Technical Report

UCAM-CL-TR-648
ISSN 1476-2986

Number 648

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Cassandra: flexible trust management and its application to electronic health records

Moritz Y. Becker

October 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2005 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

The emergence of distributed applications operating on large-scale, heterogeneous and decentralised networks poses new and challenging problems of concern to society as a whole, in particular for data security, privacy and confidentiality. Trust management and authorisation policy languages have been proposed to address access control and authorisation in this context. Still, many key problems have remained unsolved. Existing systems are often not expressive enough, or are so expressive that access control becomes undecidable; their semantics is not formally specified; and they have not been shown to meet the requirements set by actual real-world applications.

This dissertation addresses these problems. We present CASSANDRA, a role-based language and system for expressing authorisation policy, and the results of a substantial case study, a policy for a national electronic health record (EHR) system, based on the requirements of the UK National Health Service's National Programme for Information Technology (NPfIT).

CASSANDRA policies are expressed in a language derived from Datalog with constraints. CASSANDRA supports credential-based authorisation (e.g. between administrative domains), and rules can refer to remote policies (for credential retrieval and trust negotiation). The expressiveness of the language (and its computational complexity) can be tuned by choosing an appropriate constraint domain. The language is small and has a formal semantics for both query evaluation and the access control engine.

There has been a lack of real-world examples of complex security policies: our NPfIT case study fills this gap. The resulting CASSANDRA policy (with 375 rules) demonstrates that the policy language is expressive enough for a real-world application.

We thus demonstrate that a general-purpose trust management system can be designed to be highly flexible, expressive, formally founded and meet the complex requirements of real-world applications.

Acknowledgements

First, I would like to express my gratitude to my supervisor, Peter Sewell. His excellent guidance, astute comments and critical feedback throughout the last years have greatly helped to shape this work into what it is. I thank Peter also for his constant encouragement whenever I was in doubt.

I am grateful to Trinity College for providing me with an inspiring, friendly, and, at times, exciting environment to live and work in, and for granting me a Research Scholarship and a Mini Rouse Ball Studentship. My research was funded by a Gates Cambridge Scholarship. I would like to thank the Bill and Melinda Gates Foundation for their generosity.

My thanks to Nick Gaunt from the NHS Modernisation Agency and Malcolm Oswald from NPfIT for providing me with essential documents and insider information on NPfIT. Special thanks to Rishi Mukherjee for explaining to me the paper-based patient record system at Addenbrooke's Hospital.

I owe a great debt to colleagues in and outside Cambridge: Andy Gordon, Gavin Bierman, Vicky Weissman, and Tuomas Aura for their discussions and feedback, and generally helping me to think through many of the issues of the thesis. To Scott Stoller for lecturing on CASSANDRA in his course on Security Policy Frameworks at Stony Brook University, and for pointing out several bugs.

Many thanks to my colleagues in the Computer Laboratory, especially the other PhD students in the Theory and Semantics Group, and in particular to Matthew Parkinson for all kinds of support and tips, but above all for entertaining me with time-consuming puzzles. To Gareth Stoyle for setting up LSD for Lunch, and helping me with Linux and stock purchases, and generally being a great office mate.

I also thank my friends Arne Heizmann and Christian Schreiber for reading and commenting my work, and listening to my talks. I will be forever grateful to Andrea Görtler for being my friend, but also for her proof-reading, constructive criticism and endless patience. My gratitude extends to all my other friends in Cambridge, who are a continuous source of inspiration and make my life bewilderingly colourful.

Contents

1 Introduction

1.1 Research Motivation

In July 2003, a "serious breach of patient confidentiality" was reported at the Belfast Royal Victoria Hospital. The Real IRA had gained access to electronic patient records and gathered information on police officers to target them for murder [BBC03]. This example of the potential dangers of electronic medical databases may be extreme, but there have been several cases in the recent past where lax attitudes towards patient confidentiality has led to the ruin of people's lives [And96a, And99, Bro00].

The NHS's National Programme for Information Technology (NPfIT), conceived in 2002 and to be completed by 2010, is the largest and most ambitious IT project in the UK. NPfIT's central project is the Spine, a nation-wide online electronic health record (EHR) service. There is serious concern amongst the public and health professionals alike that the Spine will erode confidentiality even further[1].

With traditional paper-based health records, abuse by outsiders requires physical access to the paper files, and the likelihood of confidentiality breaches by health professionals is minimised by social control — in any case, the potential damage is kept local. In contrast, a national EHR system inherently contains higher risks for several reasons: the aggregation and centralisation of data from millions of patients combined with the wide availability of data to network users; the lack of social control mechanisms; and the possibility of automated attacks. It is clear that any such system requires sophisticated access control technology. With today's technology, it is still a daunting task to design a clinical information system of such a scale that is efficient, easy to use, and upholds the highest security standards.

The example of online electronic health records illustrates a current research problem in information security: that of access control in widely-distributed dynamic systems such as

[1] For a selection of articles reflecting the public scepticism regarding NPfIT, see e.g. [Haw03, Rog03, CS03, Col03a, Pal03, Col03c, Cro03, Fou03, Arn03, Ley04, Col04b, Car05, Mul05, Kei05].

the Internet. In the past, access control models have been designed for relatively simple environments: single computers with multiple accounts (e.g. discretionary/mandatory access control) or centralised networks of a single enterprise (e.g. role-based access control). We are interested in distributed systems that are heterogeneous, dynamic, decentralised and large-scale, with possibly billions of autonomous entities (which may be human individuals but could also be software agents, enterprises or other administrative domains) wishing to access and share resources in a secure and controlled fashion.

What is it that makes access control in this context so very challenging? Firstly, the authorisation policies (i.e., the rules governing who can access which resources) can be extremely complex, may partially rely on and interact with policies of other users on the network, and they can frequently change. Secondly, unlike before, access control cannot be solely based on identification and authentication of individuals anymore. In the new environments, subjects wish to collaborate and share their resources with previously unknown users. We therefore need a way to establish trust between mutual strangers.

In the *trust management* approach [BFL96], entities explicitly define their authorisation policy, using a high-level *policy language* to express rules governing access to protected and shared resources. The policy can then be used by the trust management software to decide whether or not to grant access requests. This approach separates policy from the actual implementation, thus simplifying security administration and policy evolution.

Access authorisation in a trust management system is based on *credentials*, digitally signed certificates asserting attributes about entities holding them. In the simplest case, the credential may be an identity certificate, but in general the attributes need not be identity-based. Peers establish trust between each other by exchanging sets of suitable credentials. This idea is common in everyday life: for example, a passenger can check in and request a boarding pass at the airport on production of a passport and a flight ticket.

Although a fair number of trust management systems and policy languages have been proposed, the area is still immature and many key issues have remained unsolved. There is controversy about what features are needed and how expressive and generic the policy language should be. The diversity of emerging applications with differing security requirements has led to the development of policy languages supporting a wide variety of policy-related constructs, e.g. role hierarchies, delegation, appointment, or separation of duties. Often, existing languages are extended to accommodate more complex policies. For example, the role-based policy language RT_0 [LWM01] was extended to RT_1 to handle parameterised roles, and to RT^T to express separation of duties [LMW02]. Another extension of RT, RT_1^C [LM03], provides constructs for limiting the range of role parameters using constraints.

However, adding constructs to a language in an *ad hoc* fashion to increase its expressiveness has several disadvantages. Firstly, it is unlikely that the extension will cover all policies of interest; secondly, the semantics and implementations of the language have to be changed; thirdly, languages with many constructs are harder to understand and to reason about; and lastly, query evaluation usually becomes computationally more expensive with increasing expressiveness (in some cases, the language is even undecidable, so processing access requests may never terminate).

We believe that a policy language should be flexible enough to be able to express constructs and policy idioms directly without having to add them as first-class features. Furthermore, it should be possible to tailor its expressiveness and computational complexity to meet the application's requirements without having to change its semantics.

Most existing systems do not have a sound formal foundation. In many cases, only

the semantics of the policy language is formally specified. However, what is generally missing is a formal semantics for the operations of the system, specifying, for example, what happens precisely to the state of the system when a role is deactivated or when a credential is requested. Without a formal semantics we cannot be certain about the correctness of evaluation algorithms and of policies written in that language. Furthermore, it also leads to misunderstandings when communicating with other researchers in the area: Everyone uses terms such as "policy" and "role" with a different intended meaning.

At the same time, the research area suffers from a surprising lack of practical examples and case studies of real-world authorisation policies that are large and complex. Obviously, without such examples it is hardly possible to determine the general requirements of policy languages, so it is not surprising that up to now, policy languages have remained of mostly academic interest.

The NHS Spine is a much-cited example of an application that would actually require such a sophisticated policy language: The proposed system is distributed, complex, and large. Moreover, the security requirements are likely to change frequently as changes in the relevant legislation and national guidelines will need to be immediately reflected in the EHR service's policy. The confidentiality requirements are complex: For example, patients will be, to some extent, allowed to configure fine-grain access control to their own records. These requirements have so far only been specified in informal documents, incomplete and written in plain English.

The main thesis of this dissertation is that a general-purpose trust management system can be designed to be highly flexible, expressive, formally founded and meet all requirements of real-world applications.

To justify this thesis, we propose CASSANDRA, a trust management system whose expressiveness and computational complexity can be flexibly adjusted by a so-called constraint domain parameter. In combination with a suitable constraint domain, CASSANDRA's expressiveness surpasses that of existing systems, although its grammar is simple and elegant. Both language and the access control engine have a formal semantics, and we prove theorems guaranteeing termination of query processing. Furthermore, CASSANDRA can automatically retrieve missing credentials over the network and supports automated trust negotiation, a model in which credentials themselves are treated as confidential resources. This dissertation further presents the details of a substantial case study on a formal CASSANDRA policy for the NHS Spine. The case study confirms that distributed applications will indeed require policy-based trust management with a higher degree of flexibility and expressive power than offered by previous solutions.

1.2 Research Contribution

We have carefully explored the design space of trust management and policy specification and have developed CASSANDRA, a role-based trust management system with unique features.

Firstly, the expressiveness (and hence also its computational complexity) of its policy language is easily adjustable. The language is based on a clear mathematical foundation, that of Datalog with constraints, or Datalog^{C}. As such, it is parameterised by a so-called constraint domain. Depending on the application's requirements, a suitable constraint domain can be "plugged" into the system (i.e., into the formal semantics as well as into an actual implementation) as an independent module without having to change its semantics. By

factoring out the constraint domain, the language syntax and semantics are kept small and simple. In particular, CASSANDRA has no explicit provisions for standard policy idioms such as role hierarchy, separation of duties or delegation; instead, it is truly policy-neutral in that it can encode such idioms (and many variants) as well as more advanced mechanisms such as automated credential retrieval and trust negotiation.

Secondly, we present a complete formal specification for CASSANDRA. This includes not only the policy language semantics but also an operational semantics of the access control engine. The operational semantics specifies the global state changes upon a request such as deactivating a role. The system's formal foundation eliminates ambiguities and could be used for the development of policy analysis techniques.

Thirdly, we have explored the tradeoff between expressiveness and decidability. Constraint compactness [Tom97] is a sufficient logical condition on constraint domains that guarantees decidability of policy queries, irrespective of the policy and the query. We prove constraint compactness theorems for a number of interesting and useful constraint domains. However, constraint compactness is generally hard to prove and rather restrictive. We prove a soundness theorem for a static groundness analysis procedure that can be used to cut down overly expressive constraint domains to a constraint compact fragment at runtime.

Finally, we present the results of our case study on a CASSANDRA policy for a national EHR service in the UK. Our policy is based on official NHS and Department of Health (DoH) specification documents that have also been given to potential suppliers. With 375 rules, the policy is the largest and most complex real-world example we are aware of. The case study has provided valuable lessons on the requirements of a trust management system. The EHR policy has been successfully tested on our preliminary prototype implementation of CASSANDRA.

Parts of this work have been described in refereed conference papers [BS04a, BS04b], and a technical report [Bec05].

1.3 Dissertation Outline

The remainder of the dissertation is structured as follows.

- Chapter 2 reviews research related to access control in general, and the trust management and policy approach in particular.
- Chapter 3 provides an informal overview of CASSANDRA, including its architecture, its policy language and its policy enforcement mechanisms.
- The policy language is formally specified in Chapter 4. This includes the specification of its syntax and a fixed-point semantics, as well as formal definitions for the notion of policy query and answer.
- Chapter 5 discusses computability and decidability issues with respect to policy query evaluation. It is shown that constraint compactness is a sufficient condition for decidability. The chapter also describes a number of interesting constraint domains along with algorithms for solving constraints and proofs for constraint compactness.
- Authorisation decisions are made by querying the policy. Chapter 6 describes an efficient distributed query evaluation algorithm that is based on resolution with memoing, to guarantee termination with constraint compact constraint domains. We also prove that the algorithm can be modified to perform groundness analysis on policies,

and how groundness can ensure termination even with an overly expressive constraint domain.

- Chapter 7 defines the operational semantics of CASSANDRA's access control engine. We present formal transition rules that specify the conditions and the resulting global state changes of performing an action on a resource, activating a role, deactivating a role, and requesting a credential.
- Chapter 8 shows how standard policy idioms can be encoded in CASSANDRA, including role hierarchies, separation of duties, delegation and appointment, and trust negotiation strategies.
- Chapter 9 presents the results of our case study on a UK electronic health record service. We first give some background information and an overview of the current situation concerning the NHS's plans to introduce an nation-wide EHR service. Then we describe our CASSANDRA policy for the service in detail and illustrate the complexity of the policy requirements involved with a scenario.
- Chapter 10 briefly describes our (so far incomplete) prototype implementation of the system.
- Chapter 11 concludes this dissertation with a discussion of CASSANDRA's design and the lessons learnt from the case study, a summary of the main contributions, and an outline of our future work.

Chapters 5 and 6 are mainly for the mathematically inclined reader and can be safely skipped without significant loss of continuity.

2 Related Work

Trust management is essentially access control in a widely-distributed environment where authorisation cannot be based on identity authentication. In this chapter, we first give some background information on access control, explain why mandatory and discretionary access control models are insufficient for large organisations and how this led to the development of role-based access control. This is followed by a review of major work in the field of trust management.

2.1 Access Control Models

2.1.1 Mandatory Access Control

Mandatory Access Control (MAC) refers to policies that are enforced independently of users' discretions or actions. The prime example of MAC is the *Multi-Level Security* policy used in military environments, first formalised by Bell and LaPadula [BL75] and restated in [San93]. In this model, access to objects is based firstly on their sensitivity or classification, represented by a security label, and secondly on the clearance of the user, also represented by a security label. These labels form a lattice; for example, an often used set of labels is *unclassified* $\leq$ *confidential* $\leq$ *secret* $\leq$ *top-secret*. Policy enforcement is then based on two principles that force information flow to be unidirectional:

- The *no read-up* principle states that a subject has read access to an object only if the clearance of the subject is greater than or equal to the classification of the object.
- The *no write-down* principle allows a subject to write to an object only if the subject's clearance is less than or equal to the classification of the object.

The second principle prevents users or programs from accidentally or maliciously declassifying sensitive information. MAC was designed for military systems but is not well suited for most other purposes, including commercial organisations. Having only a fixed set of

security labels and two simple access rules, MAC policies are usually too restrictive and rigid.

2.1.2 Discretionary Access Control

Discretionary Access Control (DAC) is based on the idea that users are owners of objects and are trusted to manage permissions to access the objects they own. The control is discretionary in the sense that users are allowed to grant or deny access to objects they control to other users at their own discretion. Users can also delegate control over objects to other users.

A DAC policy is based on the identity of users and objects and can be represented by an *access control matrix* [Lam71] where users are represented on the rows, and objects along the columns. Each entry contains the permission mode (e.g. read or write) of the user with respect to the corresponding object.

The notion of access control matrix is a useful abstraction for representing DAC policies. It is, however, in general too large to be stored and managed conveniently. Real-world implementations exploit the fact that the matrix is usually very sparse, and there are various ways of representing it more efficiently. *Access control lists* (ACL) represent each column of the matrix as a list, hence each object is associated with the set of authorised users. A similar scheme is used in the UNIX file system, where each file is associated with access modes for the file owner, group, and everyone else in the system. *Capability-based access control* stores the matrix by rows. Here, users are associated with the objects they are authorised to access.

DAC policies are hard to administer, especially when the system is large and dynamic. Adding and removing users or objects can be expensive, depending on whether ACLs or capabilities are used in the system. For example, removing an object from a capability-based system involves traversing each user's capability list.

2.1.3 Role-Based Access Control

Both MAC and DAC models are not sufficient for large commercial organisations [CW87]: MAC is clearly too rigid, and DAC is hard to administer. In most organisations, users act in the capacity of a role or a job function, and access control decisions are determined by the responsibilities and privileges associated with a role. This observation has led to the development of *role-based access control* (RBAC).

Many different RBAC models have been proposed over the last decade (e.g. [FK92, NO93, SCFY96, NO99, SFK00, FSG+01]), but most researchers would agree on RBAC_0 [SCFY96] as the core model. The key components in RBAC_0 are sets of users, roles and permissions. The policy is then specified by a user assignment relation, associating users to roles they hold, and a permission assignment relation linking roles to the granted permissions. To support the principle of least privilege [SS75, CW87], which states that only the minimal required set of permissions should be available to a user in a given context, roles are activated within sessions. A user acquires only the permissions of the activated roles. A user can activate multiple roles in a single session, and act in multiple sessions at the same time.

The RBAC96 [SCFY96] family of RBAC models also defines a number of extensions of the basic model. RBAC_1 extends RBAC_0 with role hierarchies by introducing a seniority order between roles. The permissions of a role are inherited by holders of more senior

roles. However, there are also other variants and interpretations of role hierarchy [FK92, FB97, San98, Mof98].

The $RBAC_2$ model extends $RBAC_0$ with constraints on the relations, and $RBAC_3$ combines constraints with role hierarchies. Several languages [CS95, AS00, RZFG01] have been invented for expressing various kinds of RBAC constraints, and separation of duties constraints in particular [Kuh97, GGF98, AS99]. Again, it has been observed that many different variants of separation of duties exist [SZ97], the crudest classification being static versus dynamic separation of duties.

Other RBAC-related policy idioms include role delegation and revocation (again with many subtle variants) [NC00, BS00, ZAC03], and RBAC administration using RBAC itself [SBC+97]. Parameterised roles and permissions [GI97, LS97] have been introduced to provide a finer granularity of control and to decrease the number of roles.

RBAC is often said to be policy-neutral in the sense that many policies, including DAC and MAC, can be expressed with roles. The notion of roles as an indirection between users and permissions makes RBAC scalable and greatly simplifies policy administration in large organisations with a high turnover of users. Another way to look at roles is to view them as attributes of users [LMW02]. In contrast to DAC and MAC, authorisation in RBAC is therefore based on user attributes rather than on identity. This point of view is especially useful in the context of trust management, where users are initially unknown and authorisation cannot be based on identity.

2.2 Trust Management

This section first outlines the main aspects of the design space of trust management. This is followed by a review of major work in the field of trust management and distributed access control systems.

The notion of trust[1] has been increasingly viewed as an important concept in information security since the last decade. Blaze, Feigenbaum and Laze were the first to introduce the term "trust management" in [BFL96], arguing that authorisation mechanisms in a distributed system should support decentralisation of administrative tasks, expressive and extensible policies, and local policies. Authorisation is based on digital credentials, and policies are written in a policy language. A query evaluation engine checks whether a set of credentials proves that a request complies with the local policy.

Trust management is a relatively new area of research, but a number of systems with differing design features have been proposed. In the following, we introduce some terminology used in describing the design space of trust management systems.

Expressive Power Security policies of enterprises and organisations usually exist only in the minds of security administrators, or, at best, in the form of informal natural-language documents describing security goals.

[1] There is no consensus in the literature on the definition of trust in the context of authorisation, and how it should be computed. For example, some authors have suggested that the amount of trust between two subjects can change over time and is a function of initial trust, past interactions and other incomplete information. They model trust not as a binary predicate but rather as a set of ordered values such as the real numbers [JT99, NK03, CNS03]. For example, an unknown subject would initially be assigned the lowest trust value. In reputation-based systems (e.g. [ARH97, ST03]) the trust value assigned to a subject also depends on the subject's reputation, i.e. the trustworthiness as perceived and reported by other users. For a clarification of trust and related concepts, see [GS00, MMH02].

In the trust management approach, such informal policies are translated into *policy rules*[2] written in a formal *policy language*. A formal policy language that aspires to capture informally stated security rules must obviously be sufficiently *expressive*.

The expressiveness of a language can be measured by its ability to express standard *policy idioms* that are thought to be frequently required in practice. Often-cited policy idioms are role hierarchy, separation of duties, role delegation, role appointment, revocation, cardinality constraints and temporal constraints.

Negation is used to state negative conditions (e.g. in separation of duties), as well as to express *prohibition* rules. If a system allows both prohibitions and permission statements, it requires some form of *conflict resolution* procedure. Negation in the presence of recursion is problematic as it can easily lead to semantic ambiguity, and higher computational complexity (or even undecidability).

In order to express a policy idiom, a language can provide a special-purpose *language construct* for the idiom. To support role hierarchies, for example, a language could provide the hierarchy relation as a special construct, and include it in its language semantics. Alternatively, a language could *encode* the policy idiom directly in the language, just using the more general and basic constructs. The advantage of the latter approach is that it can express different variants of a single idiom. The former approach, on the other hand, facilitates more concise and perhaps more readable policies — however, in the second approach, concise special constructs could also be introduced as "macros" standing for the actual construct encoding.

Distributed Credential Management In trust management systems, authorisation is based on *credentials*. Credentials are digital certificates signed and vouched for by an *issuer*, and contain an assertion, usually about a particular *subject*. In the very simplest case, the assertion could be a binding between a subject's name and a public key; such credentials are known as *identity certificates*. In the systems we are considering, the contents of a credential are usually more complex, for example a predicate or even a policy rule.

Credentials are usually submitted by the requester along with a request. The system then decides whether the request is granted, based on the submitted credentials and the local policy. Systems that support *automated credential retrieval* can retrieve missing credentials over the network on behalf of the requester. This requires the system to be able to find out at which *location* the credential is stored. So far, it has been assumed in the literature that credentials are stored either with the issuer or with the subject.

Credentials may contain statements that are confidential themselves, and must therefore be protected. An entity may be willing to disclose a particular credential only when the other party has revealed some of their own credentials, which in turn may again be protected. In general, there will be several stages of credential exchanges until a sufficiently high level of mutual trust has been established. This kind of credential protection is called *automated trust negotiation* and is a very new area of research. Automated trust negotiation requires some form of meta-policy to specify credential protection.

Scalability and Complexity A trust management system should be *scalable*: it should be easy to write concise and human-readable policies for very large environments, and to update policies in highly dynamic environments. *Roles* known from RBAC, and in particular

[2] We use the term "policy" to refer to the set of policy rules pertaining to an entity. In the literature, "policy" can sometimes also refer to a single policy rule.

parameterised roles, can be used to keep policies concise and easy to administer.

Furthermore, query evaluation and access-control decisions should always be fast, even in large and complex applications. This condition depends mainly on the computational complexity of the policy language. It is intuitively clear that high expressiveness generally comes along with higher complexity; and conversely, a tractable language may come at the cost of limited expressiveness. Whilst tractability is certainly a desirable property, even a language with an intractable theoretical worst-case complexity may perform well in practice. In general, how expressive — and conversely, how efficient — a system must be will depend on the actual application.

In any case, it is strongly desirable that a policy language be *decidable*. If a language is made too expressive, it may become Turing-complete, in which case query evaluation may never terminate.

2.2.1 PolicyMaker/KeyNote

PolicyMaker [BFL96, BFK99c] was the first example of a trust-management system. Its credentials and policies are fully programmable in that the choice of the policy language is left open. This makes policy compliance checking undecidable in general, but in the special case of monotonic assertions the problem is polynomial-time solvable. PolicyMaker was designed to be minimal and analysable, so a large amount of responsibility is placed on the calling application, including policy enforcement, cryptographic verification and credential gathering.

KeyNote [BFK99a, BFK99b], PolicyMaker's successor, differs from PolicyMaker in that cryptographic verification is performed by the trust management system and policies are written in a specific assertion language, but the core concepts are the same. Applications receiving authorisation requests call the KeyNote policy engine to check whether the request should be authorised. They provide the engine with the public key of the requester along with local policies, credentials (policies signed by foreign parties) and an *action attribute set* (a list of attribute/value pairs) containing relevant information. The policy engine replies with a string; in the simplest case, this may be "grant" or "deny". It is up to the application to gather the credentials and to construct an appropriate action attribute set that reflects the security requirements of the request. It is also the responsibility of the application to interpret the string returned by KeyNote and to enforce the policy accordingly.

KeyNote credentials and policies essentially specify delegation of authority to a set of public keys, conditioned on *assertions*, boolean tests on the action attributes written in the KeyNote assertion language. The assertion language provides string comparisons, regular expressions and arithmetic operations and comparisons.

2.2.2 SPKI/SDSI

The *Simple Public Key Infrastructure/Simple Distributed Security Infrastructure* (SPKI/SDSI) is a project being developed by the IETF SPKI Working Group, whose goal is to develop Internet standards for certificate formats, key acquisition protocols, and other authorisation operations. Originally, SDSI [RL96, Aba98] and SPKI were two separate projects motivated by the inadequacy of global name schemes as used in the X.509 public key infrastructure [ITU00]. Later, both merged into a single framework referred to as SPKI/SDSI 2.0 or just SPKI [Ell99, EFL+99].

Name certificates have the purpose of defining local names representing a set of public keys. Formally, a name certificate contains a four-tuple (K, A, S, V) where K is the public

key of the issuer, A is the local name defined by this certificate, S is another name or a public key, and V is a validity specification. This tuple defines the local name A in the sense that all keys represented by the subject S belong to K's A. Names defined thus can be used as subject names in other certificates. For example, if K_A is Alice's public key and K_{UCAM} is the public key of the University of Cambridge, the name certificate

(K_A, `Universities`, K_{UCAM}, 1 year)

certifies that the key K_{UCAM} belongs to Alice's local definition of "Universities".

SPKI/SDSI introduces the notions of linked local names which can be interpreted as groups of entities or public keys. A name in the local name space of the user Alice (with public key K_A) is either of the form $K_A A_1$ referring to all keys belonging to the name A_1 as defined locally by Alice, or can be a linked name of the form $K_A A_1 A_2 ... A_n$ referring to all keys belonging to the name A_n as recursively defined by any key belonging to $K_A A_1 ... A_{n-1}$.

For example, K_A `Universities` may refer to all universities in Alice's local name space, and K_A `Universities Students` would refer to all students in the local name space of those universities that are defined in Alice's local name space. The naming mechanism facilitates the use of local aliases (e.g. `Universities`) and the linking of several local name spaces. Thus it is possible to use a group of entities as a single subject where the actual members of the group may be unknown to the local site; in other words, the naming authority may be delegated to remote sites. This is something that simpler systems such as KeyNote cannot express.

Policies are represented by the certificates in the network. There are two kinds of certificates, name certificates and authorisation certificates.

Authorisation certificates can be represented by a five-tuple (K, S, d, T, V) where K is again the issuer's public key, S is a local name under K's authority representing the subject of authorisation, d a boolean delegation flag indicating whether the authorisation can be delegated, T specifies the specific permission to be granted, and V is a validity specification. For example, the five-tuple

(K_A, (K_A `Universities Students`), `false`, `use_discount`, 1 year)

grants a discount to all students of universities recognised by K_A. Note that the authority to identify students has been delegated to the universities.

An entity is permitted to access a resource if there is a chain of name and authorisation certificates that delegates the requested privilege to the entity's public key. [CEE+01] presents an algorithm based on rewriting rules for deducing such a certificate chain, given a collection of certificates. The algorithm does not deal with the problem of automatic certificate retrieval.

2.2.3 RT

Li, Mitchell and Winsborough argue in [LMW02] that authorisation in collaborative environments should be based on authenticated attributes of the entities rather than on public keys. They propose a family of attribute-based access control (ABAC) languages called RT. RT can be seen as a combination of RBAC, SDSI's linked name scheme and Li's Delegation Logic [LGF03]. This combination enables RT to overcome the failure of key-based trust management languages (e.g. KeyNote or the early SPKI) to express statements such as *"anyone who is a student is entitled to a discount"*.

The policy rules, or "credentials", as they are called in RT, specify only the role membership relation. In their simplest language, RT_0 [LWM01], policy rules can be of four different forms:

- $A.r \leftarrow B$ means "A says that B is a member of role r"
- $A.r_0 \leftarrow B.r_1$ means "A says that X is a member of role r_0 if B says that X is a member of role r_1". This provides direct delegation of role membership defining authority.
- $A.r_0 \leftarrow A.r_1.r_2$ means "A says that X is a member of role r_0 if a member of role r_1 says that X is a member of r_2. This feature represents attribute-based (as opposed to identity-based) delegation of authority.
- $A.r_0 \leftarrow B_1.r_1 \cap ... \cap B_n.r_n$ means "A says that X is a member of r_0 if, for $1 \leq i \leq n$, B_i says that X is a member of r_i. This type of credential introduces conjunction of role membership conditions.

RT_1 introduces parameterised roles; RT_2 adds to RT_1 constructs for grouping logically related objects such as resources or permissions. RT^T adds a construct called "manifold roles" for expressing threshold and separation of duty policies. RT^D supports delegation of role activations.

The credentials in these languages can be translated into Datalog. In RT's youngest offspring, RT_1^C [LM03], credentials are translated into Datalog with constraints. However, their use of constraints is rather limited: constraints are used only to define a range on each role parameter; constraints between two parameters are not permitted in order to keep policies more comprehensible and to guarantee tractability. The language also does not allow constraints on the role names or the entities of a credential.

The problem of automatically constructing an RT credential chain is addressed in [LWM01]. The problem of credential discovery is central to trust management systems which are based on the notion of delegation. The paper presents an algorithm for solving the problem without the often-made assumption that credentials are either stored centrally or that they are exclusively stored with the issuer. It is suggested that many situations require more flexible credential storage policies where some credentials are stored with the issuer and some with the subject. However, the situation where a credential is neither stored with the issuer nor with the subject is not considered.

RT also introduces *application domain specification documents* (ADSD): an ADSD defines a common vocabulary for role names, data types and role parameters and provides natural language descriptions of the components. Credentials can refer to a specific ADSD to declare the vocabulary it is using.

2.2.4 QCM/SD3

QCM [GJ00b] and its successor SD3 [Jim01] are trust management systems for building secure name servers, public key directories and distributed repositories of security policies.

QCM's policy language uses set comprehensions to define sets of expressions that can be queried for membership. Credentials are signed documents asserting membership information about sets. Similar to SPKI/SDSI and RT, set names can be prefixed with a public key expression to indicate delegation of authority to that key. For example,

$$AliceKeys = \{k \mid (\text{"Alice"}, k) \in K_{bob}\$PKD\}$$

is a policy specifying Alice's keys by delegating authority to Bob's definition for the set PKD. When QCM is given this policy along with a credential

$$K_{bob} \text{ says } (\text{“Alice”}, K_{alice}) \in PKD,$$

it can evaluate *AliceKeys* to $\{K_{alice}\}$.

QCM was the first trust management system to consider automated credential retrieval. In the example above, if QCM had not been given the credential, the system would attempt to retrieve it from Bob's server. This generally requires credentials to be signed online, which can be prohibitively expensive. QCM also has an offline signing mode where only pre-signed credentials are used to compute answers. To make denial-of-service attacks less likely QCM can be switched into a verify-only mode which verifies policies on the basis of locally available credentials only.

In [GJ00a], an extension to QCM is introduced to deal with credential revocation in a uniform framework. The distribution of revocation information uses the same mechanisms as the ones used for normal credentials. To avoid non-monotonicity and logical inconsistencies, variables and names are tagged with polarities (+/-). A polarity type system (positive names for set membership, negative names for non-membership) ensures monotonicity in the sense that approximate query results are lower bounds for membership queries and upper bounds for non-membership queries. The polarity discipline also ensures that policies use a set name either exclusively for testing membership or for testing non-membership.

SD3 [Jim01] extends QCM with recursion. The language is based on distributed Datalog, where predicates can be prefixed with an entity's public key and its IP address. The example from above could be expressed in SD3 as

$$\mathsf{aliceKey}(k) : -K_{bob}\,@\,IP_{bob}\,\$\,\mathsf{pkd}(\texttt{Alice}, k)$$

SD3 also allows intensional answers. So if the evaluation engine is queried with the predicate $\mathsf{aliceKey}(k)$ without any supporting credentials, it could either contact Bob's server or it could return the above rule as an intensional answer. [JS01] discusses the exact meaning of a correct and complete answer in the presence of intensional answers, and describes an algorithm for query evaluation.

The query evaluation engine produces a formal proof for the correctness of the answers. The proof is then checked by a simple proof checker that is much smaller than the policy evaluation engine, thus dramatically reducing the trusted computing base.

Both QCM and SD3 are similar to PolicyMaker and KeyNote in that they do not have an access control engine: it is the application's job to formulate appropriate queries and to interpret and enforce the answers.

2.2.5 Binder

Binder [DeT02] is another example of a Datalog-based policy language. Authorisation statements are expressed by defining arbitrary predicates. The declarations can be simple facts such as

```
can(Alice,Read,File123).
employee(Alice,Heffers).
```

or rules such as

$$\mathsf{can}(x, \texttt{Read}, f) :- \mathsf{employee}(x, \texttt{Heffers}).$$

Binder supports delegation of authority and certificate-based authorisation in a similar way as SD3. Predicates appearing in the body of a rule can be tagged by a so-called context, a public-key that indicates who vouches for the predicate. The context can also be a variable of sort "public-key". For example, the following rule delegates authority over the employee predicate to the HR department:

$$\begin{aligned}&\mathsf{employee}(x, y) :- \\ &\quad z \text{ says } \mathsf{employee}(x, y), \\ &\quad \mathsf{hr\text{-}key}(z)\end{aligned}$$

A tagged predicate K says P can be satisfied by an imported certificate asserting the fact P and signed by K. However, Binder certificates can also assert rules, as opposed to only facts as in SD3. In general, an imported certificate signed by K is interpreted by tagging the head of the rule and all body predicates without context tags with the context "K says". For example, a certificate asserting the rule above, signed by Heffers and imported into a new context would be interpreted as

$$\begin{aligned}&\texttt{Heffers} \text{ says } \mathsf{employee}(x, y) :- \\ &\quad z \text{ says } \mathsf{employee}(x, y), \\ &\quad \texttt{Heffers} \text{ says } \mathsf{hr\text{-}key}(z)\end{aligned}$$

Authorisation to access a resource is checked by deducing a predicate from the collection of local policy and imported certificates. The exact form of the query depends on the application. Binder does not specify any special predicates with a predefined authorisation semantics, and cannot express policies with state.

By regarding predicate contexts as extra predicate arguments, Binder policies can easily be fully translated into Datalog. Policy queries are therefore polynomial-time decidable.

2.2.6 Lithium

Halpern and Weissman have studied authorisation policy specification and analysis using various fragments of first-order logic. The most expressive, still tractable, fragment in [HW03] is referred to as Lithium in [WL04].

In [HW03], policy rules of the following form are considered:

$$\forall x_1, .., x_m.\, f \Rightarrow (\neg)\mathsf{Permitted}(e, a)$$

where f is any first-order formula and $(\neg)$Permitted indicates that the predicate may or may not be negated. Permitted(e, a) is interpreted as "entity e has permission to perform action a". Similarly, $\neg$Permitted(e, a) means "e is forbidden to do a". Furthermore, the context in which rules are used are specified by a so-called environment, a collection of formulas not containing the Permitted predicate.

Deducing if an action is permitted (or explicitly forbidden) amounts to checking whether the corresponding Permitted (or $\neg$Permitted) predicate is logically entailed by the conjunction of all rules and the environment. Similarly, policy consistency reduces to logical satisfiability of the conjunction of all rules and the environment.

The main difference to the Datalog-based systems is that Lithium allows explicit prohibition. In contrast to Datalog with negation, where a predicate is assumed to be false if it cannot be proved (closed world assumption), a predicate in Lithium is false only if its negation can be logically deduced. It is argued that prohibition is necessary for detecting conflicts in merging two policies.

As the general validity problem for first-order logic is undecidable, a variety of syntactic restrictions are considered that lead to decidability. In essence, rules are restricted such that the formula f above is a quantifier-free conjunction of possibly negated predicates. To achieve tractability in Lithium, further restrictions are placed on the use of the equality relation and predicates appearing on both sides of an implication.

2.2.7 OASIS

The *Open Architecture for Secure Interworking Services* (OASIS) is a role-based trust management framework [HBM98, YMB02, BMY02]. It extends RBAC with parameterised roles and privileges, environmental predicates that are dependent on external run-time information and the notion of *role appointment*. Many RBAC models support role delegation. However, in real life there is more often the need to assign permissions not held by the assigner herself. For example, a hospital receptionist may assign the permission to read a patient's record to the patient without being permitted to read the record herself. Appointment can thus be seen as a generalisation of delegation. In OASIS, appointments are granted by appointment certificates.

OASIS also supports immediate and cascading role revocation, even across the network. This is achieved by revocation notifications sent through an asynchronous event infrastructure [BMB+00]. Revocation between distributed OASIS services as well as role and policy component sharing are governed by bilateral service level agreements (SLA). Automatic SLA generation for OASIS is studied in [Bel04].

There are two kinds of policy rules: authorisation rules assign privileges to roles, and activation rules specify prerequisites for role activation. Authorisation rules are of the following form:

$$r, e_1, ..., e_n \vdash p$$

This rule assigns the privilege p to the role r if the environmental predicates e_1,...,e_n hold.

Role activation rules are of the following form:

$$r_1, ..., r_k, ac_1, ..., ac_\ell, e_1, ..., e_m \vdash r$$

The target role r can be activated if the prerequisite roles r_i have been activated, the appointment certificates ac_i are present and not revoked, and the environmental predicates e_i hold. Prerequisite conditions can be tagged to indicate that they are membership conditions. As soon as a membership condition ceases to hold, the target role is revoked. Since the target role can itself be a membership condition for other roles, this can trigger cascading revocation.

2.2.8 Automated Trust Negotiation

Some trust management systems address the problem of automated credential retrieval, i.e., how and where to find credentials on behalf of users to satisfy a policy. A related

problem is that of *automated trust negotiation* (ATN), the process of finding, retrieving and exchanging credentials that may be protected resources themselves.

In [WSJ00], a model is presented for exchange strategies of potentially sensitive attribute-based credentials. Both client and server credentials are protected by Credential Access Policies (CAP). A CAP specifies which credentials the other party must present in order for the protected credential to be disclosed. Typically, a client approaches a server with a service request along with a set of supporting credentials. If the submitted credentials are not sufficient, the service will reply with its service-governing policy, asking for necessary client credentials. Since these credentials may be sensitive, the client may wish to see specific server credentials before disclosing the requested credentials. In short, trust is established between client and server through incremental exchange of sensitive credentials.

Two different negotiation strategies are discussed. Both strategies are sound, complete and terminating even in the case of cyclic dependencies. The first one is the Eager Strategy where at each step, each party discloses all credentials that are currently unlocked until the original request can be satisfied or no new credentials are unlocked. This strategy is simple and fairly efficient but has the drawback that more credentials are disclosed than necessary. With the Parsimonious Strategy, on the other hand, each credential request is derived from the preceding request and the local CAP. This ensures that the set of credentials exchanged is kept locally minimal. The drawback is that the negotiation process may leak information about possession of credentials without disclosing the credentials themselves.

This information leakage problem is addressed in [WL02]. The behaviour of an entity during the negotiation process generally reveals information about credential possession. The paper proposes to specify an acknowledgement policy on top of the authorisation policy to protect possession information about credentials. The negotiating partner first has to satisfy the acknowledgement policy if she asks for an acknowledgement-protected credential. This should ensure that entities behave uniformly no matter whether a credential is actually in possession of the entity or not. The problem is hard and only partially solved in [WL02].

In [YWS01], it is argued that entities should be given the freedom to choose their own credential disclosure strategy and be able to change their strategy even during negotiation, as long as the strategies on both sides remain interoperable. Two negotiation strategies are said to be interoperable if trust negotiation using these two strategies succeeds whenever it is possible. A family of negotiation strategies, Disclosure Tree Strategies, is presented which are all mutually interoperable.

3

Design Issues and Overview

As a trust management system with a high-level policy language, CASSANDRA's purpose is to enable potentially large networks of entities to share their resources under well-defined restrictions, specified by local authorisation policies, even if they are mutual strangers. CASSANDRA was designed to be a system to meet the complex requirements of real-world applications. To satisfy these requirements, our main design goals were as follows:

- **Simplicity:** The policy language should contain just a small number of basic, orthogonal first-class constructs, from which more complex features can be constructed. This approach enhances expressiveness and keeps the semantic description of the language concise. As is the case with other language design issues, this design goal involves a tradeoff: the language must still be high-level enough so that policy writing is reasonably simple and policies are comprehensible.
- **Flexibility:** Applications have widely varying needs, and it is unlikely that a single language could fulfil the entire range of application requirements. Moreover, as there is always a tradeoff between expressiveness and computational complexity, a policy language should not be more expressive than it needs to be. Ideally, it should be possible to flexibly tune the expressiveness of the language according to the application's specific needs.
- **Support for delegation of authority:** As collaborating entities may be mutual strangers, authorisation must be based on attributes (cf. [LMW02]) rather than on identity. Such attributes are certified by digitally signed credentials that can form chains of trust. The trust management system should support credentials requests, submissions, retrieval and negotiation. The policy language must be expressive enough to specify these operations.
- **Scalability:** Roles are a useful concept for simplifying policy administration. Rather than assigning permissions to individuals, rules can separately specify the role membership conditions and the role-permission relation.

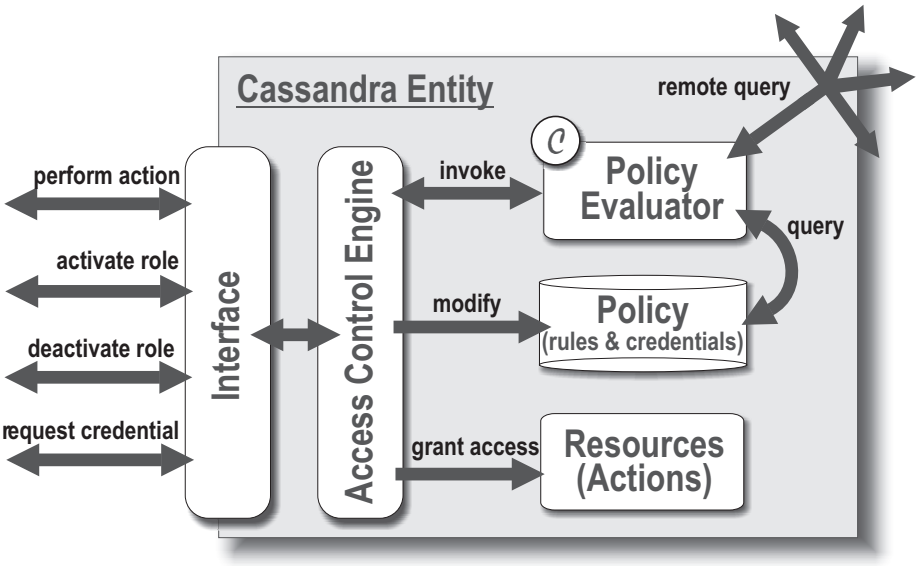

Figure 3.1: Cassandra components.

- **State awareness:** Access control decisions often depend on past events, or the system's state. For example, *"a clinician may only access a patient's data if the patient has given express consent"*, or *"a person may authorise a payment if they were not its initiator"*. The system's model should be explicit about the state and the state changes that are relevant to access control.

This chapter describes how these objectives are met in CASSANDRA, and provides a brief and informal overview accompanying the more detailed and technical subsequent chapters.

3.1 Architectural Overview

Imagine a network of entities[1] who would like to collaborate with each other. Every entity runs its own copy of a CASSANDRA service acting as a protective layer around the resources. A service can conceptually be broken up into several components.

Figure 3.1 shows the internal components of a CASSANDRA service. Entities can interact with each other only via the interface. The design goal of the interface was simplicity, orthogonality and generality. Consequently, the interface defines only the most essential and basic requests relevant to role-based trust management: performing an action (i.e. accessing a resource), activating and deactivating a role, and requesting a credential that perhaps is needed to gain authorisation somewhere else.

The conditions on which such requests are granted are specified by a local CASSANDRA policy. Upon a request, the *access control engine* constructs and sends corresponding policy queries to the *evaluation engine*. The evaluation engine is designed in such a way that a constraint solving module can be plugged into it as an independent module, matching the chosen constraint domain $\mathcal{C}$ utilised by the policy.

[1] *Entities* can be human users, organisations, or programs, often also called "principals" or "subjects" in the literature.

CASSANDRA supports distributed delegation of authority: a policy rule can have a condition asking for a credential signed by some other entity. Rules can also specify where such credentials are located. Query evaluation may thus trigger a series of credential requests over the network before returning with an answer. Finally, the answer is used by the access control engine to decide whether the request is to be granted.

3.2 Interface and Access Control Engine

Entities interact with each other by sending requests through the interface depicted in Figure 3.1. The requester may submit a set of credentials to the service that may support the request. For example, to receive a discount, the requester may need to submit a student's ID.

Upon a request, the access control engine invokes the evaluation engine to query the policy, checking whether the request should be granted. An authorisation decision thus amounts to a logical deduction. The exact query depends on the type of request: performing an action, activating a role, deactivating a role, or requesting a credential. With just these four basic operations we can encode more complex policy processes.

3.2.1 Performing an Action

Actions represent accesses to protected resources and are parameterised for greater scalability, e.g. `Read-file`(*file*). So if a user Sarah wants to read the file Readme, say, the corresponding query would be:

> *Can the predicate* permits(`Sarah`, `Read-file`(`Readme`))[2] *be deduced from the service's policy and the set of submitted credentials?*

Only if the evaluation engine returns a positive answer, the access control engine will authorise the request.

3.2.2 Activating a Role

Role-based access control (RBAC) was introduced to simplify security administration of large enterprises (see §2.1.3). CASSANDRA combines trust management with the notion of role. Our roles, just like the actions, are parameterised (cf. [GI97, LS97]) to enhance language expressiveness, e.g. `Student`(*age, college, subject*). Role parameters can also dramatically reduce the number of required roles in a policy.

In RBAC, members of a role can activate the role within a so-called session to obtain the permissions associated with it. This helps to uphold the principle of least privilege, as only those roles that are required for the current task need to be activated, thus reducing the chance of error. Furthermore, constraints can be placed on the roles that are simultaneously activated to express separation-of-duties constraints. For example, a doctor may be registered on an electronic health record system as both a clinician and as a patient, but the system may force the doctor to activate only one of these login roles at any time.

[2] For the remainder of the dissertation, variables will be written in small letters and italics (e.g. *file*), generic (meta) constants in italics but capitalised (e.g. some entity *E*), and concrete constants in typewriter font (e.g. `Read-file`). Predicate and function names will be written in sans serif font (e.g. the *special* predicate name permits).

In CASSANDRA, role membership is described by the canActivate predicate, and current role activations by the hasActivated predicate. A typical role activation request would be processed by the access control engine as follows:

> *Can the special predicate* hasActivated(`Sarah`, `Student`(22, `Trinity`, `Maths`)) *be deduced? If yes, then the user already has activated that role, otherwise check if* canActivate(`Sarah`, `Student`(22, `Trinity`, `Maths`)) *can be deduced.*

If the answer is positive, the request is granted. Moreover, the service "remembers" the new role activation by adding the fact

> hasActivated(`Sarah`, `Student`(22, `Trinity`, `Maths`))

to the policy.

This last point is crucial: CASSANDRA is aware of role activations (and deactivations), and hence state-dependent policy rules can be expressed as conditions on role activations. The two state-dependent examples from page 30 could then be expressed as

> *A clinician may only access a patient's data if the patient has activated the express-consent role*

and

> *A person may authorise a payment if they have not activated the payment's initiator role.*

As can be seen from these examples, the notion of role in CASSANDRA is much more general than the organisational job roles typically used in RBAC. A role can more generally be seen as an attribute that can be queried by the policy, and role activation facilitates the setting of that attribute. Activating a role can thus model a state-changing action such as giving consent or initiating a payment. The difference to simply performing an action is that activating (or deactivating) a role changes the state in a way that is directly observable within our operational model (see Chapter 7).

CASSANDRA policies can also access stateful information that is "outside" the policy and not observable within the operational semantics, via constraints that depend on the environment, e.g. the current time or data from an external database. The example above could alternatively be implemented as

> *A clinician may only access a patient's data if the external patient consent database has an entry for the patient*

and

> *A person may authorise a payment if they have not been registered as the payment's initiator in the external database.*

Policy authors are thus given the choice of which part of the system state should be kept inside the policy and encoded as role activations. In the policy rules presented in Chapters 8 and 9, we chose to encode those actions that directly influence the authorisation dynamics of the application, e.g. consent management, personal access restrictions, or delegation of permissions.

3.2.3 Deactivating a Role

Deactivating a role can be used in different contexts. If the role is a job role, deactivating one's own role may model logging off. If the role is someone else's job role, it corresponds to role revocation. If the role is used to signify the setting of an attribute (see §3.2.2), deactivation can be used to unset the attribute, e.g. to withdraw previously given consent.

If for example some user Tim requests to deactivate Sarah's student role, the access control engine would issue the following series of queries:

> *Can* hasActivated(`Sarah`, `Student`(22, `Trinity`, `Maths`)) *be deduced? If this is not the case, then the role can obviously not be deactivated. Otherwise check if* canDeactivate(`Tim`, `Sarah`, `Student`(22, `Trinity`, `Maths`)) *can be deduced.*

If the request is granted, the obvious consequence is to remove the hasActivated fact from the service's policy. But CASSANDRA also supports *cascading revocation*, where one deactivation may trigger other deactivations. This form of cascading revocation is especially important in the context of role delegation and appointment. The special predicate isDeactivated indicates which further deactivations are triggered.

So the service would also issue the query

> *Given the policy, the submitted credentials and the fact* isDeactivated(`Sarah`, `Student`(22, `Trinity`, `Maths`)), *for which values of "user" and "role" can the predicate* isDeactivated(*user*, *role*) *be deduced?*

Cascading revocation is then performed by additionally removing all facts

> hasActivated(*user*, *role*)

from the policy, for all computed values of *user* and *role*.

3.2.4 Requesting a Credential

A policy may require the presence of credentials for granting a request. A rule can specify that a credential should be automatically fetched over the network by the service. Alternatively, it could require that a credential must be submitted locally to support the request, perhaps to save bandwidth or simply because the service does not know where the credential can be retrieved.

Since requests can be supported by the submission of credentials, there must be a way for entities to acquire credentials. Credentials generalise digital certificates, where a name is bound to a public key and thereby an entity's identity is asserted. In CASSANDRA, a credential can assert any predicate and is always signed by some entity. For example, if isStudent is a user-defined predicate name, then

> `UCam`.isStudent(`Sarah`, `Trinity`, `Maths`)

would denote a credential signed by the University of Cambridge.

Requesting a credential thus amounts to asking a service for a signed assertion of a predicate. The matter is slightly complicated by the fact that in general predicates may contain sensitive information and therefore must themselves be protected. The special predicate canReqCred specifies who and under which conditions a predicate may be disclosed as a credential.

If Tim requests the above credential from UCam, UCam's access control engine will first issue the query

> *Can* canReqCred(Tim, UCam.isStudent(Sarah, Trinity, Maths)) *be deduced from* UCam*'s policy?*

Alternatively, Tim could also send a credential request leaving some of the parameters unspecified, for example leading to the query

> *Can* canReqCred(Tim, UCam.isStudent(*student*, *college*, *subject*)) *be deduced?*

The answer to this query may be a constraint on the parameters such as "*college*=Trinity, *subject*=Maths", which could be interpreted as *"Tim is only allowed to request student credentials about Trinity Maths students"*. This answer is then used by the access control engine to issue a new query

> *Can* isStudent(*student*, *college*, *subject*) *be deduced, where college*=Trinity *and subject*=Maths*?*

Suppose the answer to this query is "*student* $\in$ {Sarah, Jo, Jenny}", in which case Tim is sent a credential with all the predicate parameters correspondingly constrained or instantiated. Tim can now use this credential to support his requests at other services.

We have seen above that the four kinds of requests are decided by the access control engine issuing corresponding queries of the form "*can P be deduced from the policy?*", where *P* is a predicate involving one of the special predicates names permits, canActivate, hasActivated, canDeactivate, isDeactivated and canReqCred. Next, we outline the policy language used to define the conditions under which such predicates are true. The access control engine is described in full detail in Chapter 7.

3.3 Policy Specification and Evaluation

When writing policies, it is natural to think in terms of the goals rather than sequences of operations. That is why the declarative paradigm known from logic programming lends itself so well to policy specification. In logic programming, rules are written to define predicates, and the programmer need not specify explicitly the order in which the rules are applied, so the logic is effectively separated from the control.

Consequently, many policy languages are based on (negation-free) Datalog [AHV95]. Datalog rules are essentially Horn clauses without function symbols[3] of the form

$$p_0(\vec{x_0}) \leftarrow p_1(\vec{x_1}), \ldots, p_n(\vec{x_n}).$$

The translation of informal policies into policy rules is then relatively straightforward since most rules are naturally expressed as conditional sentences. Datalog's recursive nature makes it also well-suited for transitive policy idioms such as delegation or appointment.

However, the lack of function symbols is often too restrictive in practice. Many policy languages extend Datalog to increase its expressiveness (cf. Chapter 2). In contrast, one of our design requirements was to allow the language's expressiveness to be flexibly tuned: CASSANDRA is based on Datalog with Constraints, or Datalog$^{\mathcal{C}}$, and is thus parameterised by a so-called constraint domain with which the language can be flexibly tailored to the

[3] This restriction makes the language computable (in contrast to e.g. Prolog) and thus suitable for deductive database queries.

application's specific needs. In Datalog^C, rules can express constraints (taken from the constraint domain) on predicate arguments, for example

$$\begin{aligned}&\mathsf{isMatureStudent}(name) \leftarrow\\ &\quad \mathsf{isStudent}(name, college, subject),\\ &\quad \mathsf{matriculated}(name, matricDate),\\ &\quad \mathsf{dateOfBirth}(name, birthDate),\\ &\quad matricDate - birthDate \geq 21 \text{ years}\end{aligned}$$

3.3.1 Special Predicates

Policy rules can refer to arbitrary, user-defined predicate names such as isMatureStudent or dateOfBirth. Apart from these, there are also six *special predicates* that we have already encountered in §3.2. These are special in the sense that they are used by the access control engine to make authorisation decisions (and possibly to modify the policy state) for the four types of requests: performing an action, activating and deactivating a role, and requesting a credential.

1. $\mathsf{permits}(e, a)$ indicates that the entity e is permitted to perform action a.
2. $\mathsf{canActivate}(e, r)$ indicates that the entity e can activate role r.
3. $\mathsf{hasActivated}(e, r)$ indicates that the entity e has currently activated role r.
4. $\mathsf{canDeactivate}(e_1, e_2, r)$ indicates that e_1 can deactivate e_2's role r (if e_2 has really currently activated r).
5. $\mathsf{isDeactivated}(e, r)$ indicates that e's role r shall be deactivated as a consequence of another role deactivation (if e has really currently activated r).
6. $\mathsf{canReqCred}(e_1, e_2.p(\vec{e}))$ indicates that e_1 is allowed to request and receive credentials asserting $p(\vec{e})$ and issued by e_2.

Recall from §3.2.2 that if an entity activates a role, a new hasActivated statement is added to the policy; and conversely, the statement is removed from the policy when the role is deactivated. It may come as a surprise that role activation and deactivation requests modify the policy. Indeed, in a real implementation, hasActivated predicates would likely be stored separately from the policy for efficiency reasons. However, reflecting the current role activations directly in the policy simplifies our model, as the entire state of the system (i.e., which roles are activated) is then captured by the policy alone.

Moreover, the conditions in a rule are sometimes concerned with whether somebody *has* activated a role, and sometimes whether somebody *can* activate a role (i.e., is a member of the role). Similarly, as we shall see in §3.3.3, credentials are just signed predicates, and are sometimes used for asserting that somebody *has* activated a role, and sometimes that somebody *can* activate a role. Therefore, expressing both role activation and role membership in form of predicates is a logical design decision that keeps the model uniform.

The following examples of policy rules illustrate the use of these special predicates[4]. This rule specifies that human resource managers who have activated their role can register new

[4]These are deliberately simple examples. More "impressive" examples are found in Chapters 8 and 9.

employees in any department, apart from executive board members.

$$\begin{aligned}&\mathsf{permits}(e, \texttt{Register-employee}(\mathit{name}, \mathit{dept})) \leftarrow \\ &\quad \mathsf{hasActivated}(e, \texttt{Manager}(\texttt{HR})), \\ &\quad \mathit{dept} \neq \texttt{executive-board}\end{aligned}$$

The next policy fragment consists of three rules. The first rule allows Alice and Bob to activate the administrator role if their user roles have been activated; the second one allows everyone to deactivate their own user role; and the third rule specifies that an administrator role is automatically deactivated if the corresponding user role is deactivated.

$$\begin{aligned}&\mathsf{canActivate}(e, \texttt{Admin}()) \leftarrow \\ &\quad \mathsf{hasActivated}(e, \texttt{User}()), \\ &\quad e \in \{\texttt{Alice}, \texttt{Bob}\}\end{aligned} \tag{3.1}$$

$$\mathsf{canDeactivate}(e_1, e_2, \texttt{User}()) \leftarrow e_1 = e_2 \tag{3.2}$$

$$\begin{aligned}&\mathsf{isDeactivated}(e, \texttt{Admin}()) \leftarrow \\ &\quad \mathsf{isDeactivated}(e, \texttt{User}())\end{aligned} \tag{3.3}$$

Note that the second rule could have been written more concisely as

$$\mathsf{canDeactivate}(e, e, \texttt{User}()).$$

The first argument of every special predicate is an explicit *subject* parameter; for example, in the case of canDeactivate, the first parameter specifies *who* can perform the deactivation. In the simple rules shown above, the subject in the head is always the same as in the body predicates. However, there are rules where this is not the case. These rules cannot be easily expressed in languages where the subjects of head and body are implicitly the same.

3.3.2 Aggregation

The CASSANDRA policy language also allows a restricted form of aggregation, a powerful higher-order construct for computing the set of all different values that satisfy a predicate. An *aggregation rule* is a rule whose head predicate contains one of the two aggregation operators, group or count. Intuitively, the group operator finds the set of all different ground values that satisfy a predicate, whereas count computes the size of that set. For example, consider the following rule:

$$\begin{aligned}&\mathsf{fndMonkeys}(\mathsf{group}\langle x \rangle, \mathit{age}) \leftarrow \\ &\quad \mathsf{hasActivated}(x, \texttt{Monkey}(\mathit{age}))\end{aligned}$$

Querying the policy with the query $\mathsf{fndMonkeys}(\mathit{monkeys}, 5)$ would find the set *monkeys* of all entities M that satisfy the predicate

$$\mathsf{hasActivated}(M, \texttt{Monkey}(5)),$$

that is, the set of all active monkeys of age 5. Similarly, the query cntMonkeys$(n,5)$ on the rule below would find the number n of active monkeys of age 5.

$$\begin{aligned}&\mathsf{cntMonkeys}(\mathsf{count}\langle x\rangle, age) \leftarrow\\ &\quad \mathsf{hasActivated}(x, \texttt{Monkey}(age))\end{aligned}$$

3.3.3 Credentials and Distributed Policies

The concept of delegation of authority is central in the trust management approach. A digital credential asserting a specific statement can be used for authorisation if the issuer is trusted to have authority over the assertion. CASSANDRA can not only specify credentials as authorisation conditions but also the credential retrieval and exchange (trust negotiation) process.

In CASSANDRA, the properties asserted by credentials are constrained predicates. Furthermore, a credential is signed and issued by an entity called *issuer*, and is held by an entity called *location*. We write

$$\begin{aligned}&\texttt{Sarah}\Diamond\texttt{UCam}.\mathsf{canActivate}(\texttt{Sarah}, \texttt{Student}(age, college, subject)) \leftarrow\\ &\quad age = 22,\\ &\quad college = \texttt{Trinity},\\ &\quad subject \in \{\texttt{Maths}, \texttt{Spanish}\}\end{aligned}$$

to represent a credential held by Sarah and issued by the University of Cambridge (expressed by the prefix "Sarah◊UCam."), and asserting that Sarah is a 22-year old Maths and Spanish student at Trinity College.

In CASSANDRA, every predicate in a rule can have a location prefix and an issuer prefix. This enables rules to refer to predicates that are stored somewhere else on the network and are signed by someone else. In other words, rules can refer to credentials. As in the examples given in the previous sections, we usually omit those prefixes that refer to the location of the rule itself. Example 3.3 written out with all prefixes would have read (assuming that it is stored in E's policy):

$$\begin{aligned}&E\Diamond E.\mathsf{isDeactivated}(e, \texttt{Admin}()) \leftarrow\\ &\quad E\Diamond E.\mathsf{isDeactivated}(e, \texttt{User}())\end{aligned}$$

The following example illustrates how CASSANDRA deals with predicates that refer to remote locations. The first rule belongs to the (fictitious) policy of Heffers online bookshop, which offers discounts to Philosophy, Maths and Computer Science students from UCam and the Anglia Polytechnic University (APU). According to that rule, if a user sends a request to Heffers to activate a DiscountUser role, the bookshop's CASSANDRA service will send a credential request back to the user, asking for an appropriate Student credential issued by either UCam or APU:

$$\begin{aligned}&\texttt{Heffers}\Diamond\mathsf{canActivate}(requester, \texttt{DiscountUser}()) \leftarrow\\ &\quad requester\Diamond issuer.\mathsf{canActivate}(requester, \texttt{Student}(age, college, subject)),\\ &\quad subject \in \{\texttt{Philosophy}, \texttt{Maths}, \texttt{CompSci}\},\\ &\quad (issuer = \texttt{UCam} \vee issuer = \texttt{APU})\end{aligned}$$

Suppose Sarah's CASSANDRA service is willing to return a copy of the requested student credential to Heffers because she does indeed possess a suitable student credential from

UCam (Sarah reads Maths and Spanish, by her first rule below), and because her policy also includes a rule stating that she is willing to disclose her student credential to any requester (Sarah's second rule below):

$$\begin{aligned}(1)\ & \texttt{Sarah}\Diamond\texttt{UCam}.\mathsf{canActivate}(\texttt{Sarah}, \texttt{Student}(\mathit{age}, \mathit{college}, \mathit{subject})) \leftarrow \\ & \quad \mathit{age} = 22, \\ & \quad \mathit{college} = \texttt{Trinity}, \\ & \quad \mathit{subject} \in \{\texttt{Maths}, \texttt{Spanish}\}\end{aligned}$$

$$\begin{aligned}(2)\ & \texttt{Sarah}\Diamond\texttt{Sarah}.\mathsf{canReqCred}(\mathit{requester}, \\ & \qquad \texttt{UCam}.\mathsf{canActivate}(\texttt{Student}(\mathit{age}, \mathit{college}, \mathit{subject}))) \leftarrow\end{aligned}$$

A detailed treatment of the policy language will be given in Chapter 4.

3.4 A Scenario

Recall that all granted requests (apart from performing an action) have side-effects on the global state, i.e., the access control engine modifies the policy. Activating a role adds a hasActivated fact (i.e., a rule without body predicates) to the service's policy; deactivating a role removes one or more hasActivated facts; and requesting a credential adds a fact representing the credential to the requester's policy.

Now that we have given an overview of both the access control engine and the language, the following example will illustrate the interplay between the two; this scenario will be revisited in more formal detail in §7.6.

Consider the three policy rules from examples 3.1, 3.2 and 3.3. Suppose the policy also contains a fourth rule indicating that Alice has activated her User role:

$$\mathsf{hasActivated}(\texttt{Alice}, \texttt{User}()) \leftarrow \tag{3.4}$$

Now suppose Alice requests to have the Admin() role activated for her. The access control engine will first query the policy with the predicate hasActivated(Alice,Admin()) to check whether Alice has already activated that role. The answer is negative, as expected, so the second query is canActivate(Alice,Admin()). This succeeds because of the first and the fourth rule, and consequently a fifth rule is added to the policy:

$$\mathsf{hasActivated}(\texttt{Alice}, \texttt{Admin}()) \leftarrow \tag{3.5}$$

Now suppose Alice requests that her User() role be deactivated. The access control engine first tries to deduce the predicate hasActivated(Alice,User()); and indeed, Alice is currently active in the user role, according to the fourth rule. Then, the policy is queried with canDeactivate(Alice,Alice,User()) which also succeeds because of the second rule. Finally, the access control engine will attempt to find all values satisfying the predicate isDeactivated(e,r) under the assumption isDeactivated(Alice,User()). By assumption, this will of course be the pair (Alice,User()) itself, and by the third rule, also the pair (Alice,Admin()). The found values match the hasActivated facts from the fourth and the fifth rules. Consequently, these two rules are removed from the policy, and we are left with the original rules 3.1, 3.2 and 3.3.

4
Policy Specification

In CASSANDRA, every entity can specify its own authorisation policy by writing a set of rules in the policy language. This chapter formally specifies the syntax and semantics of the language. We define the generic notion of constraint domain in §4.1. The syntax of roles and actions is defined in §4.2 and predicates in §4.3. These components are used to write policy rules, the syntax of which is defined in §4.4. A formal semantics of the language, based on fixed points, is specified in §4.5.

4.1 Datalog^C and Constraint Domain

Many existing policy languages are based on Datalog [AHV95], i.e., Horn clauses without function symbols, because it is a rule-based, declarative language, widely understood, and queries always terminate. However, standard Datalog is not very expressive, so many systems extend the language with *ad hoc* features. We take a different approach and base our language on an extension of recursive Datalog^C in which the expressiveness – and conversely, the computational complexity – is parameterised on the chosen *constraint domain* C [JM94, JMMS98, Rev02]. The language can thus be adapted to a wide range of applications without having to change its base semantics.

A Datalog^C rule is of the form

$$p_0(\vec{e}_0) \leftarrow p_1(\vec{e}_1), .., p_n(\vec{e}_n), c$$

where the p_i are predicate names and the $\vec{e}_i$ are (possibly empty) expression tuples (that may contain variables) matching the parameter types of the predicate. The constraint c is a constraint from the chosen constraint domain C. Intuitively, to deduce p_0, all body predicates $p_1, ..., p_n$ must be deducible in such a way that the constraint is also satisfied. A set of Datalog^C rules can then be interpreted as the deductive closure of the set.

The expressiveness of Datalog$^{\mathcal{C}}$ depends on the chosen constraint domain $\mathcal{C}$. For example, the least expressive constraint domain is the one where the only atomic constraints are equalities between variables and constants. Choosing this trivial constraint domain reduces the expressiveness of the language to standard Datalog or Horn clauses without function symbols. More powerful constraint domains often include boolean, arithmetic and set constraints, and make use of more complex expressions such as tuples, set expressions and function applications (e.g. to access the current time or other environmental information).

The computational complexity of evaluating Datalog$^{\mathcal{C}}$ programs increases with increasing expressiveness: with set constraints it is already possible to encode the Hamiltonian cycle problem, and thus all NP-complete problems. Care must be taken not to choose a constraint domain that is too expressive as this can result in programs in which queries are undecidable. We will later introduce the notion of *constraint compactness* to restrict constraint domains to those that guarantee termination of queries.

In order to be interoperable, the services on the network must agree on a common constraint domain $\mathcal{C}$. Concrete examples of useful constraint domains for authorisation policies are given in Chapter 5. Here, we formally define our notion of constraint domain.

Definition 4.1.1. (**constraint domain**) *A* constraint domain $\mathcal{C}$ *is a set of constraints, i.e. first order formulas, on elements of a domain, the set of* $\mathcal{C}$-expressions. $\mathcal{C}$ *is equipped with an interpretation that defines validity of constraints. Constraint domains must satisfy a number of minimal requirements and support certain computable operations on constraints, as set out below in §4.1.1 and §4.1.2.*

Before we specify the minimal requirements and supported operations, we first have to define the notion of satisfiability of constraints with respect to the constraint domain's interpretation.

Definition 4.1.2. (**validity, satisfiability, free variables**) *Let c be a* $\mathcal{C}$*-constraint. We write* $\models c$ *if c is valid under* $\mathcal{C}$*'s interpretation. We say a substitution* θ satisfies *c (*$\theta \models c$*) if* $\theta(c)$ *is valid. A constraint c is* satisfiable *if there is a substitution that satisfies c. The set of free variables of a constraint c is denoted by* $Fv(c)$.

4.1.1 Minimal Requirements

In the context of our policy language, constraints are conditions on predicate parameters. The set of $\mathcal{C}$-expressions must therefore at least contain all elements in *Variables* and *Entities*, fixed infinite sets of variables and entities.

For the constraints in $\mathcal{C}$ we require the following:

1. The boolean constants true and false are in $\mathcal{C}$.
2. $\mathcal{C}$ contains all equality constraints $e_1 = e_2$ for all $\mathcal{C}$-expressions e_1, e_2.
3. Closure under variable renaming: if $c \in \mathcal{C}$ then $c[x/y] \in \mathcal{C}$, for all variables x and y.
4. Closure under conjunction: if $c_1, c_2 \in \mathcal{C}$ then $c_1 \wedge c_2 \in \mathcal{C}$.
5. Closure under existential quantifier elimination: if $c \in \mathcal{C}$ and x is a variable free in c, then there are quantifier-free constraints $c_1, \ldots, c_n \in \mathcal{C}$ such that $\bigvee_{i=1}^{n} c_i$ is in disjunctive normal form (DNF) and equivalent to $\exists x(c)$, i.e.,
   - c_i is disjunction free, for all $i \in \{1, .., n\}$.
   - $x \notin Fv(c_i)$, for all $i \in \{1, .., n\}$.

- $\theta \models c_i$ for some $i \in \{1,..,n\}$ iff there exists a $\mathcal{C}$-expression e such that $\theta \models c[e/x]$.

The boolean constants, equality constraints and conjunction must be interpreted as usual:

- true is valid and false is not valid.
- If both c_1 and c_2 are valid then so is $c_1 \wedge c_2$.
- If e_1 and e_2 are syntactically identical then $e_1 = e_2$ is valid.

We will often use a comma (,) instead of $\wedge$ for conjunction of constraints.

4.1.2 Operations

We further require that $\mathcal{C}$ is equipped with three computable operations, namely *satisfiability checking*, *subsumption checking* and *existential quantifier elimination*. The rationale behind these requirements is that CASSANDRA's query evaluation algorithm and the access control engine can use these operations as an abstract interface to the constraint domain, without needing to know any particulars about the concrete constraint domain. Constraint domains can thus be "plugged" into the system as independent modules.

Definition 4.1.3. (**subsumption**) *Let c_1, c_2 be $\mathcal{C}$-constraints. Then c_1 is* subsumed *by c_2 if for all substitutions θ,*

$\theta \models c_1$ *implies* $\theta \models c_2$.

Intuitively, c_1 is subsumed by c_2 if c_1 "implies" c_2. For example,

- false is subsumed by every constraint.
- Every constraint is subsumed by true.
- $c_1 \wedge c_2$ is subsumed by both c_1 and c_2.
- $x < 0$ is subsumed by $x < 1$.
- $x < y < z$ is subsumed by $x + 1 < z$.

Definition 4.1.4. ($\Rightarrow^{\mathcal{C}}$) *Every constraint domain $\mathcal{C}$ must be equipped with a computable* subsumption checking *relation $\Rightarrow^{\mathcal{C}}$ (or simply $\Rightarrow$ if $\mathcal{C}$ is clear from the context). The relation must satisfy the following property for all constraints $c_1, c_2 \in \mathcal{C}$:*

$c_1 \Rightarrow^{\mathcal{C}} c_2$ *implies that c_1 is subsumed by c_2.*

Note that the definition of $\Rightarrow^{\mathcal{C}}$ requires the operation to be only an approximation of the "real" subsumption relation. The reason is that subsumption is used for the sole purpose of keeping the model of a Datalog$^{\mathcal{C}}$ program finite (see §4.1.3). A less exact subsumption checking relation does not affect the soundness of an implementation, but it may cause a worse termination behaviour. The subsumption checking algorithm described in 5.2.3, for example, is only approximate but strong enough to guarantee termination.

Definition 4.1.5. ($\exists^{\mathcal{C}}$) *Every constraint domain $\mathcal{C}$ must be equipped with a computable* existential quantifier elimination operation $\exists^{\mathcal{C}} : \mathcal{P}_{fin}(\mathit{Variables}) \rightarrow \mathcal{C} \rightarrow \mathcal{P}_{fin}(\mathcal{C})$ *(again, the superscript $\mathcal{C}$ is usually omitted). It takes a finite set of variables $x_1, ..., x_m$ and a constraint*

c, and returns a finite set of disjunction-free and quantifier-free constraints $\{c_1, ..., c_n\}$ *such that*

$$\bigvee_{i=1}^{n} c_i \text{ is equivalent to } \exists x_1 ... x_n(c),$$

as defined in §4.1.1.

If x is a variable, we write $\exists^{\mathcal{C}} x(c)$ *as shorthand for* $\exists^{\mathcal{C}}\{x\}(c)$*. If V is a set of variables, we write* $\exists^{\mathcal{C}}_{-V}(c)$ *as shorthand for* projecting *c* onto *V*, *i.e.*, $\exists^{\mathcal{C}}(Fv(c) - V)\ (c)$.

For example,

- $\exists x(x = y) = \{\mathsf{true}\}$
- $\exists x(x = 3 \wedge y < x) = \{y < 3\}$
- $\exists x(x < y \wedge y < z) = \{y < z\}$
- $\exists y(x < y \wedge y < z) = \{x + 1 < z\}$
- $\exists x(\mathrm{signum}(x) = y) = \{y = -1,\ y = 0,\ y = 1\}$

4.1.3 Datalog$^{\mathcal{C}}$ Semantics

The semantics, or the *model*, of a Datalog$^{\mathcal{C}}$ program $\mathcal{P}$ can be defined as the set of all facts that can be deduced from it. This set is the least fixed point of a *consequence operator* $T^{\mathcal{D}}_{\mathcal{P}}$, a function that takes a set of facts (ground predicates) I and returns the set of facts immediately deducible from those facts together with the rules of the program $\mathcal{P}$.

In this section we assume w.l.o.g. that programs are *rectified*, i.e., in each rule, all predicate parameters are distinct variables. Furthermore, we will identify rules up to (non-clashing) variable renaming. Then $T^{\mathcal{D}}_{\mathcal{P}}$ can be defined as follows:

$$\begin{array}{lll} T^{\mathcal{D}}_{\mathcal{P}}(I) = & \{\, p(\vec{e}) \mid & \text{there is a rule } p(\vec{x}) \leftarrow p_1(\vec{x}_1), ..., p_n(\vec{x}_n), c \ \in \mathcal{P} \\ & & \text{and a substitution } \theta \text{ such that} \\ & & \bullet\ \theta \models c, \\ & & \bullet\ \theta\vec{x} = \vec{e}, \\ & & \bullet\ p_i(\theta\vec{x}_i) \in I, \text{ for all } i \in \{1, .., n\}\ \} \end{array}$$

The set $\bigcup_{i\in\omega}(T^{\mathcal{D}}_{\mathcal{P}})^i(\emptyset)$ represents the deductive closure of the rules in $\mathcal{P}$. Consider, for example, the following program:

Example 4.1.6.

$$\begin{array}{l} p(x) \leftarrow x = 3 \\ p(x) \leftarrow p(y), x > y \end{array} \tag{4.1}$$

The ground model is computed by repeatedly applying $T^{\mathcal{D}}_{\mathcal{P}}$ until a fixed point is reached:

$$\begin{array}{rcl} T^{\mathcal{D}}_{\mathcal{P}}(\emptyset) & = & \{p(3)\} \\ (T^{\mathcal{D}}_{\mathcal{P}})^2(\emptyset) & = & \{p(3), p(4), p(5), p(6), ...\} \\ (T^{\mathcal{D}}_{\mathcal{P}})^3(\emptyset) & = & (T^{\mathcal{D}}_{\mathcal{P}})^2(\emptyset) \end{array}$$

The fixed point is obtained after the second application. The problem with $T^{\mathcal{D}}_{\mathcal{P}}$ is that the ground model is often infinite, as in this example. As we are interested in computability

and practical applicability, a more useful consequence operator would be one that operates on constrained facts rather than ground facts to keep the fixed point finite [GL91, GDL95, Tom97, JMMS98]. Such an operator $T_{\mathcal{P}}^{\mathcal{C}}$ can be defined as follows:

$$
\begin{aligned}
T_{\mathcal{P}}^{\mathcal{C}}(I) = \{\, p(\vec{x}) \leftarrow c_0 \mid\ & \text{there is a rule } p(\vec{x}) \leftarrow P_1, ..., P_n, c \in \mathcal{P} \\
& \text{and constraints } c_1, ..., c_n \text{ such that} \\
& \bullet\ c_0 \in \exists_{-\vec{x}}(c \wedge c_1 \wedge ... \wedge c_n), \\
& \bullet\ c_0 \text{ is satisfiable,} \\
& \bullet\ c_i \text{ is a } \textit{contribution} \text{ from } P_i, \text{ for all } i \in \{1, .., n\}; \\
& \text{and if } p(\vec{x}) \leftarrow c_0' \in I \text{ such that } c_0 \Rightarrow c_0' \text{ then } c_0 = c_0' \ \}
\end{aligned}
$$

where c_i is a *contribution* from P_i if $P_i \leftarrow c_i \in I$.

$T_{\mathcal{P}}^{\mathcal{C}}(I)$ produces new constrained facts from a rule by finding already existing solutions in I (contributions) for each body predicate, and projecting the conjunction of all these constraints and the rule constraint onto the free variables of the rule's head predicate. The last line of the definition ensures that new constrained facts are only produced if they are not already subsumed by some existing fact.

Applying $T_{\mathcal{P}}^{\mathcal{C}}$ to the program from Example 4.1.6 yields:

$$
\begin{aligned}
T_{\mathcal{P}}^{\mathcal{C}}(\emptyset) &= \{p(x) \leftarrow x = 3\} \\
(T_{\mathcal{P}}^{\mathcal{C}})^2(\emptyset) &= \{p(x) \leftarrow x = 3,\ p(x) \leftarrow \exists y(y = 3 \wedge x > y)\} \\
&= \{p(x) \leftarrow x = 3,\ p(x) \leftarrow x > 3\} \\
(T_{\mathcal{P}}^{\mathcal{C}})^3(\emptyset) &= (T_{\mathcal{P}}^{\mathcal{C}})^2(\emptyset)
\end{aligned}
$$

The fixed point is now finite and is reached after the second application of the operator, because $p(x) \leftarrow \exists y(y > 3 \wedge x > y) \equiv p(x) \leftarrow x > 4$, which would have been added in the third step, is already subsumed by $p(x) \leftarrow x > 3$, provided that the (possibly approximate) operation $\Rightarrow$ is sufficiently exact. Without the last line in the definition of $T_{\mathcal{P}}^{\mathcal{C}}$, the operator would carry on adding redundant credentials indefinitely, and the fixed point would be infinite. The resulting fixed point is a finite representation of the infinite ground model.

Of course, the fixed point may still be infinite in general; after all, Datalog$^{\mathcal{C}}$ with a sufficiently expressive constraint domain $\mathcal{C}$ is Turing-complete. But the fixed point of $T_{\mathcal{P}}^{\mathcal{C}}$ is often finite even if the ground model is infinite.

As CASSANDRA is an extension of Datalog$^{\mathcal{C}}$, its semantics is also defined as the fixed point of a consequence operator based on $T_{\mathcal{P}}^{\mathcal{C}}$, discussed in §4.5.1.

4.2 Roles and Actions

The CASSANDRA policy language is role-based to make access control administration simpler and more scalable. Our *roles* and *actions* (generalised privileges) are parameterised for higher expressiveness [GI97, LS97]: e.g. the role `Clinician`(*org, spcty*) has parameters for the health organisation and the specialty of the clinician; the action `Read-record-item`(*pat, id*) has parameters specifying the item identifier of a patient *pat*'s health record. The syntax for roles and actions is:

Expressions	e	::=	x
		|	E
		|	$()$
		|	*role*
		|	*action*
		|	... other $\mathcal{C}$-expressions
Roles	*role*	::=	$R(e)$
Actions	*action*	::=	$A(e)$
Role Names	R	$\in$	*RoleNames*
Action Names	A	$\in$	*ActionNames*
Entities	E	$\in$	*Entities*
Variables	x	$\in$	*Variables*

RoleNames and *ActionNames* are infinite sets of role names and action names, respectively. The grammar of expressions specifies the parameters that roles and actions can take, and will generally include all $\mathcal{C}$-expressions. The empty tuple $()$ is used for parameter-less roles, actions and predicates, and has type *unit* (see below). We write $R()$ instead of $R(())$ (and similarly for actions and predicates).

A type system ensures that policies are well-formed. The type system for policy rules and their subcomponents (such as roles and actions) rely on the type system of the global constraint domain $\mathcal{C}$. In Chapter 5, we will present a number of constraint domains along with their type systems. In general, a type system will have type judgements for expressions e of the form $\Gamma \vdash e : \tau$ where τ is some type and Γ is a finite function mapping variables to types. Similarly, a type system will have type judgements for constraints c of the simple form $\Gamma \vdash c$. The types we will be considering in this and the following chapters are of the following form:

Types	τ	::=	*entity* | *const* | *int* | *role*(τ) | *action*(τ) | *unit*
		|	$\tau_1 \times ... \times \tau_n$
		|	*set*(τ)

Note that the types generally also depend on the expressions of the constraint domain.

The typing rules for the empty tuple, entities and variables are as follows:

$$\frac{}{\Gamma \vdash () : \mathit{unit}} \qquad \frac{E \in \mathit{Entities}}{\Gamma \vdash E : \mathit{entity}} \qquad \frac{x \in \mathit{Dom}(\Gamma) \quad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Role names and action names are associated with the type of the parameters they take. *Type*(R) denotes the parameter type of a role name R (and similarly for action names). The typing rules for roles and actions are as follows:

$$\frac{\Gamma \vdash e : \tau \quad \mathit{Type}(R) = \tau}{\Gamma \vdash R(e) : \mathit{role}(\tau)} \qquad \frac{\Gamma \vdash e : \tau \quad \mathit{Type}(A) = \tau}{\Gamma \vdash A(e) : \mathit{action}(\tau)}$$

4.3 Predicates

In the RBAC model, relations are used to define role membership, session activations and role permissions. In CASSANDRA, these and other access control relations are specified implicitly by rules defining six special predicates (their precise operational semantics is formalised in Chapter 7): apart from these six predicates with a special access control meaning, policy writers can introduce further auxiliary user-defined predicate names. The syntax for predicates is as follows:

$$
\begin{array}{rrcl}
\text{Predicates} & \mathit{pred} & ::= & \mathsf{permits}(e_1, e_2) \\
& & | & \mathsf{hasActivated}(e_1, e_2) \\
& & | & \mathsf{canActivate}(e_1, e_2) \\
& & | & \mathsf{canDeactivate}(e_1, e_2, e_3) \\
& & | & \mathsf{isDeactivated}(e_1, e_2) \\
& & | & \mathsf{canReqCred}(e_1, e_2.\mathit{pred}) \\
& & | & p(e_1, ..., e_n), \text{where } n \geq 0 \\
& p & \in & \mathit{UserPredNames}
\end{array}
$$

Again, we impose a type regime on predicates to enforce well-formedness. The typing rules for the six special predicates are as follows:

$$\frac{\Gamma \vdash e_1 : \mathit{entity} \qquad \Gamma \vdash e_2 : \mathit{action}(\tau)}{\Gamma \vdash \mathsf{permits}(e_1, e_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{entity} \qquad \Gamma \vdash e_2 : \mathit{role}(\tau)}{\Gamma \vdash \mathsf{hasActivated}(e_1, e_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{entity} \qquad \Gamma \vdash e_2 : \mathit{role}(\tau)}{\Gamma \vdash \mathsf{canActivate}(e_1, e_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{entity} \qquad \Gamma \vdash e_2 : \mathit{role}(\tau)}{\Gamma \vdash \mathsf{isDeactivated}(e_1, e_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{entity} \qquad \Gamma \vdash e_2 : \mathit{entity} \qquad \Gamma \vdash e_3 : \mathit{role}(\tau)}{\Gamma \vdash \mathsf{canDeactivate}(e_1, e_2, e_3)}$$

$$\frac{\Gamma \vdash e_1 : \mathit{entity} \qquad \Gamma \vdash e_2 : \mathit{entity} \qquad \Gamma \vdash \mathit{pred}}{\Gamma \vdash \mathsf{canReqCred}(e_1, e_2.\mathit{pred})}$$

User-defined predicate names are associated with the type of their parameters, denoted by $\mathit{Type}(p)$. The typing rule for user-defined predicates is:

$$\frac{p \in \mathit{UserPredNames} \qquad \mathit{Type}(p) = \tau_1 \times ... \times \tau_n \qquad \Gamma \vdash e_1 : \tau_1 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash p(e_1, ..., e_n)}$$

4.4 Rules

The policy rule language in CASSANDRA extends Datalog^C's predicates for the purpose of credential-based trust management. A credential can be seen as a constrained predicate, vouched for by an *issuing* entity and stored at a *location* entity. Correspondingly, CASSANDRA predicates in a rule are tagged with a location and an issuer, e.g. `Alice◊Bob`.$p(x)$. The location (in this example Alice) is postfixed by the symbol "◊", and the issuer (here Bob) by ".".

The notion of an issuer prefix is comparable to the linked names in the SDSI [RL96, Aba98] part of SPKI/SDSI [Ell99, EFL+99], and to the role prefixes in RT [LWM01, LMW02]. Our tagging syntax may be reminiscent of SD3 [Jim01], where predicates can also be prefixed with an a location and an issuer. However, the location in SD3 is interpreted as the IP address of the issuer, whereas in CASSANDRA, the location is semantically independent of the issuer: here, it is interpreted as the location at which the *predicate* can be deduced.

We further extend Datalog^C with *aggregation operators* that can be used, for example, for cardinality constraints and restricted forms of negation (see §4.4.1).

The syntax for policy rules is as follows:

Head	*head*	::=	$E_1 \Diamond E_2.pred$
		|	$E_1 \Diamond E_2.p(\mathsf{group}\langle x\rangle)$
		|	$E_1 \Diamond E_2.p(\mathsf{count}\langle x\rangle)$
		|	$E_1 \Diamond E_2.p(\mathsf{group}\langle x\rangle, e_1 ..., e_n)$
		|	$E_1 \Diamond E_2.p(\mathsf{count}\langle x\rangle, e_1 ..., e_n)$
Body Pred	P	::=	$e \Diamond e'.pred$
Credential	*cred*	::=	$head \leftarrow c$
Rule	*rule*	::=	*cred*
		|	$head \leftarrow P_1, ..., P_n, c$
Constraint	c	$\in$	$\mathcal{C}$

Definition 4.4.1. (head, body, location, issuer, credential, policy) *A policy rule consists of a prefixed* head *predicate, a list of prefixed* body *predicates, and a constraint from $\mathcal{C}$. The prefixes e_{loc} and e_{iss} of a predicate $e_{loc} \Diamond e_{iss}.p(\vec{e})$ are called the* location *and the* issuer of the predicate, *respectively. The location and the issuer of the head of a rule are called the* location *and the* issuer of the rule, *respectively. A rule of the form $head \leftarrow c$ is called a* credential *or* credential rule. *If a policy rule is not a credential rule, its location and issuer must be identical. Only credentials may have an issuer that differs from the location: these represent credentials issued by foreign parties. Finally, a* policy *of an entity E is a set of rules with location E.*

We will often omit the constraint of a rule if it is true. Location and issuer prefixes are also usually omitted when they are equal to the location of the rule, and the location of the rule can also be omitted if it is clear from the context. So for example, instead of writing

$$\begin{aligned} A \Diamond A.p(x) \leftarrow\ & A \Diamond A.p_1(x),\ A \Diamond B.p_2(x), \\ & B \Diamond A.p_3(x),\ B \Diamond C.p_4(x), \\ & \mathsf{true} \end{aligned}$$

we would write (if from the context it is clear that we are talking about A's policy)

$$p(x) \leftarrow\ p_1(x),\ B.p_2(x),\ B\Diamond p_3(x),\ B\Diamond C.p_4(x)$$

The intuitive meaning of this (rather unlikely) rule would be

> $p(x)$ can be deduced from A's service if we can firstly deduce $p_1(x)$ from A's policy, secondly find the credential asserting $p_2(x)$ and issued by B in A's policy, thirdly request from B a credential asserting $p_3(x)$ and issued by A, and fourthly request from B a credential asserting $p_4(x)$ and issued by C.

The following specifies the typing rules for policy rules:

$$\frac{\Gamma \vdash e : \mathit{entity} \qquad \Gamma \vdash e' : \mathit{entity} \qquad \Gamma \vdash \mathit{pred}}{\Gamma \vdash e\Diamond e'.\mathit{pred}}$$

$$\frac{\Gamma \vdash \mathit{head} \qquad \Gamma \vdash c}{\Gamma \vdash \mathit{head} \leftarrow c}$$

$$\frac{\Gamma \vdash \mathit{head} \qquad \Gamma \vdash P_1 \qquad \cdots \qquad \Gamma \vdash P_n \qquad \Gamma \vdash c}{\Gamma \vdash \mathit{head} \leftarrow P_1, ..., P_n, c}$$

The head of a rule can contain an aggregation expression as its first parameter, either group or count; these are further explained in the next section. The type system is extended with the following two rules so that aggregation expressions can be typed as if they were expressions:

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \mathsf{group}\langle x\rangle : \mathit{set}(\tau)} \qquad \frac{x \in \mathit{Dom}(\Gamma)}{\Gamma \vdash \mathsf{count}\langle x\rangle : \mathit{int}}$$

4.4.1 Aggregation Rules

Policies often require negative conditions in the premise of a rule, e.g. that an entity has not activated a particular role, or that no entity has activated the role. Datalog^C extended with negated body predicates could express the former example, but not the latter, which implicitly involves universal quantification. Instead, we introduce *aggregation operators* [MPR90, SR91, Van92, Rev02] with which both examples can be expressed. Aggregation is also useful for grouping and cardinality constraints, e.g. constraints on the set of all role activations of a particular entity, or on the number of all such activations.

An *aggregation rule* is a rule whose head contains an aggregation operator. Intuitively, the group operator finds the set of all different ground values that satisfy a predicate, whereas count computes the size of that set.

Aggregation is a very expressive mechanism, but it has to be used carefully. In our language, predicates are generally recursive and often are satisfied by an infinite number of ground values; both these properties are problematic with aggregation. To enforce a finitary semantics and to improve efficiency, we restrict aggregation rules to be aggregation-safe (in §6.3, we will lift some of these restrictions using groundness analysis):

Definition 4.4.2. (aggregation safety) *An aggregation rule is* aggregation-safe *if it is of the form*[1]

$$E\Diamond E.p_0(\mathsf{aggOp}\langle x\rangle,\vec{y}) \leftarrow E\Diamond e'.p_1(\vec{z}),c,$$

where the so-called control parameters $\vec{y}$ *may be missing. Therefore, an aggregation rule must have exactly one body predicate, and this predicate's location must be equal to the location of the rule. Furthermore,* p_1 *must be a predicate name that is only defined by credential rules in E's policy of the form* $E\Diamond E'.p_1(\vec{z}) \leftarrow \vec{z} = \vec{K}$*, where* $\vec{K}$ *are variable-free expressions. (In practice, we will restrict* p_1 *to* hasActivated*, and write policies in such a way that all* hasActivated *rules are completely grounding credential rules.) Finally, we require that the aggregation variable x as well as all* $\vec{y}$ *are in* $\vec{z}$*.*

The restriction that the body predicate of an aggregation rule must be local (its location must be equal to E) is necessary because aggregation requires complete knowledge of the predicate. Answers from remote entities are always sound but may be incomplete as they are subject to canReqCred restrictions. The restriction that the body predicate can only be satisfied with finitely many different parameters, and that x and $\vec{y}$ occur in it, ensures that aggregation is finite. In terms of negation, this corresponds to semi-positive policies, thus avoiding negation-related issues such as undecidability and semantic ambiguity.

The separation-of-duties example in Chapter 8 shows how aggregation can be used to express universally quantified negation. Note that the kind of negation we can express via aggregation can only occur in the body of a rule and never in the head. In particular, we cannot express explicit prohibitions (as in e.g. Halpern and Weissman's logic [HW03], Ponder [DDLS01, Dam02] or in FAF [JSSS01]). Rather, we make the closed world assumption and assume that everything is prohibited unless it is explicitly permitted. Allowing explicitly negative statements would add much complexity to the language; for example, it necessitates mechanisms to resolve logical conflicts.

4.5 Language Semantics

In standard Datalog^C, a predicate can be deduced if there is a rule with a matching head, such that the rule's body predicates can be deduced while satisfying the constraint of the rule. In CASSANDRA, a body predicate $B\Diamond C.p(\vec{e})$ can refer to a remote location, if B is not equal to the local entity, say A. To deduce the predicate, A will contact B over the network and delegate authority to B to deduce the predicate. Such a remote query amounts to a credential request: B will first try to deduce $B\Diamond B.\mathsf{canReqCred}(A, C.p(\vec{e}))$ before attempting to deduce the requested predicate.

To illustrate this, consider the following example where likes is some user-defined predicate. Suppose A's policy contains the rules

$$R_1 \equiv A\Diamond A.\mathsf{likes}(A,x) \leftarrow x\Diamond y.\mathsf{likes}(y,x), x \neq y$$
$$R_2 \equiv A\Diamond B.\mathsf{likes}(B,A) \leftarrow \mathsf{true}$$

(so R_2 is a foreign credential from B), and C's policy contains

$$R_3 \equiv C\Diamond D.\mathsf{likes}(D,C) \leftarrow \mathsf{true}$$

[1] We use the vector notation $\vec{e}$ as a shorthand for a list or a set of expressions. Furthermore, $\vec{z} = \vec{K}$ is shorthand for $z_1 = K_1 \wedge \ldots \wedge z_n = K_n$.

(so R_3 is a foreign credential from D). Intuitively, R_1 means: A likes an entity x if x's policy proves that some other entity y says that y likes x. More formally, we can deduce $A\Diamond A.\mathsf{likes}(A,x)$ on A if we can deduce $x\Diamond y.\mathsf{likes}(y,x)$ on the service x provided that x is not equal to y. If, in the context of evaluation, x turns out to be A, then the body becomes

$$A\Diamond y.\mathsf{likes}(y,A) \leftarrow A \neq y,$$

to be deduced locally on A, by finding a matching credential (foreign since $A \neq y$). R_2 is such a credential, thus A has proved $A\Diamond A.\mathsf{likes}(A,A)$. Otherwise, if x turns out to be different from A, say $x = C$, A automatically requests a credential from C of the form

$$C\Diamond y.\mathsf{likes}(y,C) \leftarrow C \neq y.$$

At this point, CASSANDRA's trust negotiation mechanism comes into play: C first tries to prove

$$C\Diamond C.\mathsf{canReqCred}(A, y.\mathsf{likes}(y,C)) \leftarrow C \neq y$$

on its own policy to see whether A is allowed to request such a credential. The result of this deduction is either false (A cannot request such a credential) or some constraint on the variable y, say $y \neq E$ (A is allowed to get such a credential provided $y \neq E$). In the latter case, C will then try to prove

$$C\Diamond y.\mathsf{likes}(y,C) \leftarrow C \neq y \wedge y \neq E.$$

This is satisfied by R_3, so C will reply to A with the answer $C\Diamond D.\mathsf{likes}(D,C)$, upon which A can finally prove $A\Diamond A.\mathsf{likes}(A,C)$.

4.5.1 Consequence Operator

We will now specify the language semantics formally, thereby capturing the intuition given in the previous section. The semantics of a policy is defined by the set of all credentials (since, in our language, credentials are *facts*) that can be deduced from it. Again, as in §4.1.3, this set can be computed as the least fixed point of a consequence operator. Our consequence operator $T_{\mathcal{P}}$ is more complex than T_P^C as we have to deal with predicates that may be located elsewhere and may be protected by $\mathsf{canReqCred}$ rules, so the semantics does not only depend on the local policy of a single entity. Therefore, the parameter $\mathcal{P}$ in $T_{\mathcal{P}}$ is not just a single policy but the finite union of the policies of *all* entities[2]. Given a set of credentials I, $T_{\mathcal{P}}(I)$ returns the set of all credentials that can be deduced from I and the policies in $\mathcal{P}$ in one step.

In the following we assume w.l.o.g. that all policies are *rectified*, i.e., in each policy rule, all predicate parameters and location/issuer prefixes (apart from those in the head which have to be constant) are distinct variables. Furthermore, we will identify rules up to (non-clashing) variable renaming.

Definition 4.5.1. (global policy set, consequence operator) *Let the* global policy set $\mathcal{P}$ *be the union of the policies of all entities. (The union is disjoint as the policy rules of an entity are uniquely identified by the location prefix of the rules' heads.) The* consequence operator $T_{\mathcal{P}}$ *is a function between sets of credentials I. It is defined as follows:*

[2]We are assuming that only a finite number of entities have policies.

$$
\begin{array}{l}
T_{\mathcal{P}}(I) = \\
\quad \{\, E_{loc}\Diamond E_{iss}.p(\vec{x}) \leftarrow c_0 \mid \\
\qquad \text{there is a rule } E_{loc}\Diamond E_{iss}.p(\vec{x}) \leftarrow P_1,...,P_n, c \in \mathcal{P} \\
\qquad \text{and constraints } c_1,...,c_n \text{ such that} \\
\qquad\quad \bullet\ c_0 \in \exists_{-\vec{x}}(c \wedge c_1 \wedge ... \wedge c_n), \\
\qquad\quad \bullet\ c_0 \text{ is satisfiable,} \\
\qquad\quad \bullet\ c_i \text{ is a } \textit{contribution} \text{ from } P_i, \text{ for all } i \in \{1,..,n\}; \\
\qquad \text{and if } E_{loc}\Diamond E_{iss}.p(\vec{x}) \leftarrow c'_0 \in I \text{ such that } c_0 \Rightarrow c'_0 \text{ then } c_0 = c'_0 \\
\quad \}
\end{array}
$$

where c_i is a *contribution* from $P_i \equiv y_{loc}\Diamond y_{iss}.p_i(\vec{y})$ if

- either there is a credential $E_{loc}\Diamond E'_{iss}.p_i(\vec{y}) \leftarrow c'_i \in I$ and $c_i \in \exists_{-\vec{y}}(c'_i \wedge y_{loc} = E_{loc} \wedge y_{iss} = E'_{iss})$;
- or there is an entity $E'_{loc} \neq E_{loc}$ such that
  - there is a credential $E'_{loc}\Diamond E'_{loc}.\mathsf{canReqCred}(x_e, y_{iss}.p_i(\vec{y})) \leftarrow c'_i \in I$;
  - there is a credential $E'_{loc}\Diamond E'_{iss}.p_i(\vec{y}) \leftarrow c''_i \in I$;
  - and $c_i \in \exists_{-\vec{y}}(c'_i \wedge c''_i \wedge y_{loc} = E'_{loc} \wedge y_{iss} = E'_{iss} \wedge x_e = E_{loc})$

The definition of $T_{\mathcal{P}}$ is almost identical to $T_{\mathcal{P}}$ from §4.1.3, apart from the meaning of "c_i is a contribution from P_i". Here we have two cases.

In the first case y_{loc} (the location of the P_i) is equal to E_{loc} (the location of the rule); this corresponds to the standard case: P_i is simply deduced from the local policy.

In the second case y_{loc} is equal to some $E'_{loc} \neq E_{loc}$, so it has to be deduced from E''s policy. As this amounts to a credential request, and E'_{loc}'s credentials are protected by canReqCred rules, the corresponding canReqCred predicate must also be satisfied, as well as P_i itself. The resulting contribution c_i is a combination of the constraints from the canReqCred credential (c'_i) and of the P_i credential (c''_i), the assignments for the various prefixes y_{loc} and y_{iss}, and the assignment of the canReqCred predicate subject x_e. The variable x_e may occur in c'_i but it is not in $\vec{y}$, hence it gets existential-quantifier eliminated.

As in §4.1.3, the last line in the definition of $T_{\mathcal{P}}(I)$ is very important: a credential is added to the result only if it is not already subsumed (according to the possibly approximate subsumption approximation $\Rightarrow$) by some other credential in I. This condition tries to avoid the adding of redundant credentials.

If there were no aggregation rules, we could define the semantics of $\mathcal{P}$ to simply be the least fixed point of $T_{\mathcal{P}}$, namely $\bigcup_{i\in\omega} T^i_{\mathcal{P}}(\emptyset)$. However, in the presence of aggregation rules, we first have to specify the (finite) set $Agg_{\mathcal{P}}$ of all credentials deducible from those rules, and apply $T_{\mathcal{P}}$ to it instead. The next section deals with the definition of $Agg_{\mathcal{P}}$.

4.5.2 Aggregation Semantics

We first consider the case where the head does not contain any parameters (so-called control parameters) apart from the aggregation parameter, i.e., it is a rule R of the form $E\Diamond E.p_0(\mathsf{aggOp}\langle x\rangle) \leftarrow E\Diamond e'.p_1(\vec{z}), c$. Let $\mathcal{P}$ be the global policy set. The aggregation operator aggOp can be either group or count. If S is a finite set, then let group(S) denote

simply S itself, and $\mathsf{count}(S)$ the cardinality of the set, $|S|$. We define

$$X_R = \{K_x \mid E\Diamond E'.p_1(\vec{z}) \leftarrow c_i \in \mathcal{P}, \; c_i \text{ implies } x = K_x, \; c_i \wedge c \wedge e' = E' \text{ is satisfiable } \}$$

Intuitively, X_R is the set of all constant values that x can assume to satisfy p_i and the constraint c. X_R is finite because we assume the rule to be aggregation-safe (Definition 4.4.2). Then

$$Agg(R) = \{E\Diamond E.p_0(w) \leftarrow w = \mathsf{aggOp}(X_R)\}$$

is the set of credentials deducible from the rule R.

In the presence of control parameters $\vec{y}$, the definition of Agg is a bit more complicated. Now the rule is of the form $R \equiv E\Diamond E.p_0(\mathsf{aggOp}\langle x\rangle, \vec{y}) \leftarrow E\Diamond e'.p_1(\vec{z}), c$. We define

$$Y_R = \{\vec{K}_y \mid E\Diamond E'.p_1(\vec{z}) \leftarrow c_i \in \mathcal{P}, \; c_i \text{ implies } \vec{y} = \vec{K}_y, \; c_i \wedge c \wedge e' = E' \text{ is satisfiable } \}$$

Y_R is the set of all constant values of $\vec{y}$ that can satisfy p_1 and the constraint c. Y_R is finite because the rule is aggregation-safe.

For each $\vec{K}_y \in Y_R$, we define

$$X_{\vec{K}_y} = \{K_x \mid E\Diamond E'.p_1(\vec{z}) \leftarrow c_i \in \mathcal{P}, \; c_i \text{ implies } x = K_x, \; c_i \text{ implies } \vec{y} = \vec{K}_y, \; c_i \wedge c \wedge e' = E' \text{ is satisfiable } \}$$

This is the set of all constant values that x can assume to satisfy p_1 and the constraint c when $\vec{y}$ is assigned the values $\vec{K}_y$. In the following definition of Agg, we now also have to consider the case where $\vec{y}$ is none of the values in Y_R, in which case there is no possible value for x, so w must be assigned $\mathsf{aggOp}(\emptyset)$, i.e., either $\emptyset$ or 0.

$$\begin{aligned} Agg(R) = \; & \{E\Diamond E.p_0(w,\vec{y}) \leftarrow w = \mathsf{aggOp}(X_{\vec{K}_y}) \wedge \vec{y} = \vec{K}_y \mid \vec{K}_y \in Y_R\} \cup \\ & \{E\Diamond E.p_0(w,\vec{y}) \leftarrow w = \mathsf{aggOp}(\emptyset) \wedge \textstyle\bigwedge_{\vec{K}_y \in Y_R} \vec{y} \neq \vec{K}_y\} \end{aligned}$$

Recall the example of the aggregation rule for counting active monkeys of a given age (§3.3.2). Here we have added some role activation credential rules:

$$\begin{aligned} & \mathsf{cntMonkeys}(\mathsf{count}\langle x\rangle, age) \leftarrow \\ & \quad \mathsf{hasActivated}(x, \mathtt{Monkey}(age)) \\ & \mathsf{hasActivated}(x, \mathtt{Monkey}(age)) \leftarrow x = \mathtt{Cheeta} \wedge age = 3 \\ & \mathsf{hasActivated}(x, \mathtt{Monkey}(age)) \leftarrow x = \mathtt{Louie} \wedge age = 5 \\ & \mathsf{hasActivated}(x, \mathtt{Monkey}(age)) \leftarrow x = \mathtt{Katie} \wedge age = 3 \end{aligned}$$

For this example, Y_R would be $\{3,5\}$, the set of possible ages. Then X_3 is $\{\mathtt{Cheeta}, \mathtt{Katie}\}$,

and X_5 is {Louie}. Finally, $Agg(R)$ would evaluate to

$$\{ \begin{array}{l} \mathsf{cntMonkeys}(w, age) \leftarrow w = 2 \wedge age = 3, \\ \mathsf{cntMonkeys}(w, age) \leftarrow w = 1 \wedge age = 5, \\ \mathsf{cntMonkeys}(w, age) \leftarrow w = 0 \wedge age \neq 3 \wedge age \neq 5 \} \end{array}$$

Finally, we define

$$Agg_{\mathcal{P}} = \bigcup_{R \in \mathcal{P}} Agg(R), \text{ where } R \text{ is an aggregation rule.}$$

Then $Agg_{\mathcal{P}}$ is the set of all credentials deducible from all aggregation rules in the global policy set $\mathcal{P}$.

4.5.3 Fixed-Point Semantics

Now we can put everything together and define CASSANDRA's language semantics as the least fixed point of the consequence operator, operating on the initial set of aggregation facts.

Definition 4.5.2. (semantics) *Let $\hat{T}_{\mathcal{P}}(I) = T_{\mathcal{P}}(I) \cup Agg_{\mathcal{P}}$. The (fixed-point)* semantics *of $\mathcal{P}$ is defined as*

$$Model(\mathcal{P}) \triangleq \bigcup_{i \in \omega} \hat{T}^i_{\mathcal{P}}(\emptyset).$$

By Lemma 4.5.3, the function $\hat{T}_{\mathcal{P}}(I)$ is continuous on the powerset of credentials, hence $Model(\mathcal{P})$ is its least fixed point.

Lemma 4.5.3. *$\hat{T}_{\mathcal{P}}$ is continuous on the powerset of credentials with respect to subset ordering.*

Proof. We have to show that $\hat{T}_{\mathcal{P}}$ is monotonic and that for all increasing ω-chains $I_1 \subseteq I_2 \subset \ldots$ of credential sets we have $\hat{T}_{\mathcal{P}}(\bigcup_{i \in \omega} I_i) = \bigcup_{i \in \omega} \hat{T}_{\mathcal{P}}(I_i)$.

1. Let $I \subseteq \mathcal{J}$ and a credential $E_{loc} \Diamond E_{iss}.p(\vec{x}) \leftarrow c_0 \in \hat{T}_{\mathcal{P}}(I)$. Then either this credential is in $Agg_{\mathcal{P}}$ in which case it would also be in $\hat{T}_{\mathcal{P}}(\mathcal{J})$. Otherwise, there must be a policy rule in $\mathcal{P}$ such that I contains certain credentials for each body predicate of the rule, as specified in Definition 4.5.1. Since $\mathcal{J}$ is a superset of I, all these credentials are also in $\mathcal{J}$. Therefore $E_{loc} \Diamond E_{iss}.p(\vec{x}) \leftarrow c_0 \in \hat{T}_{\mathcal{P}}(\mathcal{J})$, so $\hat{T}_{\mathcal{P}}$ is monotonic.
2. The inclusion $\bigcup_{i \in \omega} \hat{T}_{\mathcal{P}}(I_i) \subseteq \hat{T}_{\mathcal{P}}(\bigcup_{i \in \omega} I_i)$ is straightforward by monotonicity. To prove the other direction, suppose there is a credential $E_{loc} \Diamond E_{iss}.p(\vec{x}) \leftarrow c_0 \in \hat{T}_{\mathcal{P}}(\bigcup_{i \in \omega} I_i)$. Then either this credential is in $Agg_{\mathcal{P}}$ in which case it would also be in $\bigcup_{i \in \omega} \hat{T}_{\mathcal{P}}(I_i)$. Otherwise, as in the proof for monotonicity, $\bigcup_{i \in \omega} I_i$ must contain certain credentials, as specified in Definition 4.5.1. But since the I_i form an increasing chain, there must be a finite m such that all these credentials are in I_m. Hence $E_{loc} \Diamond E_{iss}.p(\vec{x}) \leftarrow c_0 \in \hat{T}_{\mathcal{P}}(I_m) \subseteq \bigcup_{i \in \omega} \hat{T}_{\mathcal{P}}(I_i)$, as required.

□

4.5.4 Queries

Access control decisions (see Chapter 7) are based on *queries* which have the same form as credentials: $E_{loc}\Diamond E_{iss}.p_0(\vec{e}_0) \leftarrow c$. Intuitively, the *answer* to a query is a set of constraints c_i such that $E_{iss}.p_0(\vec{e}_0) \leftarrow c \wedge c_i$ can be deduced from E_{loc}'s policy. For example, the query

$$\texttt{UCam}\Diamond\texttt{UCam}.\mathsf{canActivate}(x, \texttt{Student}(subj)) \leftarrow \\ subj = \texttt{Maths}$$

may return the constraints $\{x = \texttt{Alice}, x = \texttt{Bob}\}$, and the query

$$\texttt{UCam}\Diamond\texttt{UCam}.\mathsf{canActivate}(x, \texttt{Student}(subj)) \leftarrow \\ x = \texttt{Alice} \wedge subj = \texttt{Maths}$$

would simply return $\{\mathsf{true}\}$.

An answer to a query should be sound and complete. To specify this formally, we use the definition of *ground semantics*.

Definition 4.5.4. (ground semantics) *If I is a set of credentials, we write $\|I\|$ for*

$$\{\theta(P) \mid (P \leftarrow c) \in I \ \textit{and} \ \theta \models c\}.$$

The ground semantics *of $\mathcal{P}$ is defined as $\|\mathit{Model}(\mathcal{P})\|$.*

Definition 4.5.5. (query, answer) *A* query *has the same form as a credential. An* answer *to the query $P \leftarrow c$ is a finite set of disjunction-free constraints $\{c_1, ..., c_n\}$ such that for all substitutions θ satisfying c,*

$$\theta(P) \in \|\mathit{Model}(\mathcal{P})\| \Longleftrightarrow \theta \models c_1 \vee ... \vee c_n$$

5 Constraint Domains and Decidability

As in a database system, the process of query evaluation should always terminate. Policy languages must therefore not be Turing-complete, or else an undecidable policy could be written. In CASSANDRA, the policy language is parameterised on the constraint domain $\mathcal{C}$, and it turns out that decidability and query evaluation termination depend on $\mathcal{C}$. Requirements on $\mathcal{C}$ to guarantee termination, in particular, *constraint compactness*, are discussed in §5.1. Then, we describe a few concrete examples of constraint domains in §5.2, along with algorithms for the operations they are required to support, and proofs of constraint compactness. Less mathematically-inclined readers may safely skip to Chapter 7.

5.1 Decidability

The fixed-point semantics of the global policy set $\mathcal{P}$ was defined in §4.5.3. We are interested in constraint domains for which the semantics is finite, irrespective of the policies and queries: finding an answer to a query is clearly computable for a finite semantics.

Unfortunately, many interesting constraint domains do not enjoy this property. For example, the constraint domain of finite trees are the syntactic equality constraints between finite terms from a Herbrand universe (finite trees constructed from a collection of constant and function symbols). Pure Prolog can be viewed as Datalog extended with such constraints, and pure Prolog is Turing-complete, so the query problem is not decidable in general.

Similarly, a constraint domain supporting untyped tuples can also lead to undecidable policies. Suppose constraint domain consists of the syntactic equality constraints between expressions generated from a constant 0 and n-ary tuples, for $n \leq 1$. (For simplicity we include unary tuples, so $(X) \neq X$, but actually pairs alone would already be sufficient.)

Consider the policy

$$\mathsf{add}(0,y,y)$$
$$\mathsf{add}((x),y,(z)) \leftarrow \mathsf{add}(x,y,z)$$
$$\mathsf{mul}(0,y,0)$$
$$\mathsf{mul}((x),y,z)) \leftarrow \mathsf{mul}(x,y,z'),\mathsf{add}(y,z',z)$$
$$\mathsf{exp}(x,0,(0))$$
$$\mathsf{exp}(x,(y),z) \leftarrow \mathsf{exp}(x,y,z')),\mathsf{mul}(x,z',z)$$
$$\mathsf{fermat}() \leftarrow$$
$$\quad \mathsf{exp}((x),(((n'))),x'),$$
$$\quad \mathsf{exp}((y),(((n'))),y'),$$
$$\quad \mathsf{exp}((z),(((n'))),z'),$$
$$\quad \mathsf{add}(x',y',z')$$

Encoding natural numbers as $0,(0),((0)),...$, the first three pairs of rules represent addition, multiplication and exponentiation, respectively. The query $\mathsf{fermat}()$ returns true iff the diophantine equation $x^n + y^n = z^n$ has positive integer solutions for $x,y,z \geq 1$ and $n \geq 3$. We can thus encode arbitrary systems of diophantine equations. This provides a reduction to Hilbert's Tenth Problem [Hil02], which is known to be undecidable [Mat70]. In §5.2.2, we show how a type system can restrict a constraint domain with tuples in order to preserve decidability.

Toman [Tom97] gives a sufficient property of constraint domains, *constraint compactness*, for guaranteeing finiteness of the fixed-point semantics.

Definition 5.1.1. (constraint compactness) *A constraint domain $\mathcal{C}$ is* constraint compact *if*

1. *for every $\mathcal{C}$-constraint c and variable x, $\exists^{\mathcal{C}} x(c)$ does not contain any constants or variables that are not in c;*
2. *for every set of $\mathcal{C}$-constraints C with only finitely many distinct free variables and constants occurring in it, there is a finite subset $C_{fin} \subseteq C$ which* covers *C, i.e.*

$$\textit{for all } c \in C \textit{ there exists a } c' \in C_{fin} \textit{ such that } c \Rightarrow^{\mathcal{C}} c'$$

Note that constraint compactness depends on the (possibly approximate) operation $\Rightarrow^{\mathcal{C}}$. If this operation is made too imprecise, a constraint domain may cease to be constraint compact.

Essentially, if a constraint domain is constraint compact, then every infinite set of constraints (that only mentions finitely many different variables and constant symbols) contains redundancy in the sense that there is a finite subset within it that subsumes the entire set. Or in other words, the infinite set can be cut down to a finite set that contains just as much information. Since the consequence operator $T_{\mathcal{P}}$ checks for redundancy (a credential is added only if it is not subsumed by an already existing credential), the fixed point is guaranteed to be finite. The following theorem formalises this intuition.

Theorem 5.1.2. (finite semantics) *If $\mathcal{P}$ is a global policy set of $\mathcal{C}$-policies and $\mathcal{C}$ is constraint compact then Model($\mathcal{P}$) is finite.*

Proof. Suppose *Model*($\mathcal{P}$) were infinite. As there are only finitely many predicate symbols, there must exist an infinite set $C = \{P \leftarrow c_i \mid i > 0\}$ among the generated credentials, for some

prefixed predicate P. There can only be finitely many constants and variables occurring in the constrained atoms since, if $\mathcal{C}$ is constraint compact, $T_{\mathcal{P}}$ does not introduce any new constants apart from those in $\mathcal{P}$, by Definitions 4.5.1 and 5.1.1. Therefore, by constraint compactness, there must be a finite subset $C_{fin} \subset C$ such that for all $c \in C$ there is a constraint $c' \in C_{fin}$ with $c \Rightarrow^{\mathcal{C}} c'$. But this is a contradiction because all constrained atoms in C_{fin} must have been generated after a finite number of iterations, and by construction of $T_{\mathcal{P}}$, no constraint of a credential in C generated after that can be subsumed (according to $\Rightarrow^{\mathcal{C}}$) by any constraint in C_{fin}. □

5.2 Constraint-Compact Constraint Domains

This section describes several constraint domains, each one being an extension of the previous one. For each constraint domain we specify algorithms for satisfiability checking, subsumption checking and existential quantifier elimination. We also give proofs for the constraint compactness property.

5.2.1 Minimal Constraint Domain

Every constraint domain must have equality constraints between expressions and be closed under conjunction. The expressions must at least include a countably infinite set of variables and constants for entities. The smallest constraint domain that satisfies these conditions is $\mathcal{C}_{eq}$. Its syntax is as follows:

Expressions	e	$::=$	$E \mid x$
Constraints	c	$::=$	true
		$\mid$	false
		$\mid$	$e_1 = e_2$
		$\mid$	$c_1 \wedge c_2$
Entities	K	$\in$	*Entities*
Variables	x	$\in$	*Variables*

We only have one type of expressions, namely entities. A type system is therefore not required, as all constraints can be taken to be well-typed. The semantics of $\mathcal{C}_{eq}$ is straightforward and as expected:

- $\theta \models \mathsf{true}$
- $\theta \models e_1 = e_2$ if θe_1 and θe_2 are equal
- $\theta \models c_1 \wedge c_2$ if $\theta \models c_1$ and $\theta \models c_2$

Operations

A standard unification algorithm can be used to check satisfiability of $\mathcal{C}_{eq}$ constraints. A constraint c is satisfiable iff there is a most general unifier (mgu) θ that satisfies c [PW76]. We call the mgu of a satisfiable $\mathcal{C}_{eq}$ constraint its *solved form*, and false is the solved form of a unsatisfiable constraint. For example, the solved form of $x = 2 \wedge x = y$ is $\theta = [2/x, 2/y]$.

Lemma 5.2.1. *A satisfiable constraint c_1 is subsumed by a satisfiable constraint c_2 if θ_2, the solved form of c_2, is more general than θ_1, the solved form of c_1, i.e. there exists a substitution ω such that $\theta_2 \circ \omega = \theta_1$.*

Proof. First assume that c_1 is subsumed by c_2. Then all unifiers of c_1 (i.e., substitutions that satisfy c_1) are also unifiers of c_2. In particular, θ_1, the mgu of c_1 is also a unifier of c_2, so θ_2, the mgu of c_2, is more general than θ_1.

Now assume $\theta_1 = \theta_2 \circ \omega$, and suppose $\theta \models c_1$, for some θ. Then there exists some ω' such that $\theta = \theta_1 \circ \omega' = \theta_2 \circ \omega \circ \omega'$, since θ_1 is the mgu of c_1. But θ_2 is the mgu of c_2, so we also have $\theta \models c_2$. □

For example, $x = 2 \wedge y = 2$ is subsumed by $x = y$ since $[y/x]$ is more general than $[2/x, 2/y] = [y/x] \circ [2/y]$.

Lemma 5.2.2. *If c is a $\mathcal{C}_{eq}$ constraint, then $\exists^{\mathcal{C}_{eq}} x(c) = \{c_{-x}\}$, where c_{-x} is constructed from c as follows:*

1. *Remove all subconstraints of the form $x = x$; they are redundant, and removing them does not change the semantics of the constraint.*
2. *If the remaining constraint contains a subconstraint of the form $x = e$, replace all occurrences of x by e.*

Proof. Without loss of generality, assume that c does not contain redundant subconstraints of the form $x = x$.

Suppose $\theta \models c_{-x}$. By construction of c_{-x}, $c_{-x} = c[e/x]$ for some expression e (this is trivially true if x does not occur in c), hence there exists an expression e such that $\theta \models c[e/x]$.

For the other direction, suppose there is some expression e' such that $\theta \models c[e'/x]$. Either c does not contain any subconstraint of the form $x = e$ in which case $c = c_{-x}$ and x does not occur in c, so $\theta \models c_{-x}$. Otherwise, c is equivalent to $c_{-x} \wedge x = e$, hence $\theta \models (c_{-x} \wedge x = e)[e/x]$. It follows that $\theta \models c_{-x}$ since x does not occur in c_{-x}. □

For example, eliminating x from the constraint $x = y \wedge x = z$ yields $y = y \wedge y = z$, or equivalently, $y = z$.

Constraint Compactness

Now we can show that $\mathcal{C}_{eq}$ is constraint compact.

Theorem 5.2.3. *$\mathcal{C}_{eq}$ is constraint compact.*

Proof. $\exists^{\mathcal{C}_{eq}}$ clearly does not introduce any new constants or variables. For the second condition, let C be a (possibly infinite) set of $\mathcal{C}_{eq}$-constraints such that only finitely many different variables and constants occur in the constraints. Let C' be the set of solved forms of the constraints of C. Converting these constraints to their solved forms does not introduce any new constants or variables, so there are still only finitely many of them in C'. But since solved forms are just finite substitutions, there can only be finitely many different such solved forms, so C' is finite. Every constraint in C is equivalent to a solved form in C'. This implies constraint compactness. □

5.2.2 Tupling and Projection Functions

The constraint domain $\mathcal{C}_{tup}$ adds tuples and projection functions to $\mathcal{C}_{eq}$; these constructs are useful for dealing with structured role and action parameters. As was shown in §5.1, a constraint domain with untyped tuples leads to undecidability of query evaluation. A type system is introduced to circumvent this problem.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & E \\
 & & \mid & K \\
 & & \mid & x \\
 & & \mid & () \\
 & & \mid & (e_1,...,e_n), \qquad n \geq 2 \\
 & & \mid & \pi^n_k(e), \qquad n \geq 2, 1 \leq k \leq n \\
\text{Constraints} & c & ::= & \mathsf{true} \\
 & & \mid & \mathsf{false} \\
 & & \mid & e_1 = e_2 \\
 & & \mid & c_1 \wedge c_2 \\
\text{Entities} & E & \in & \textit{Entities} \\
\text{Constants} & K & \in & \textit{Constants} \\
\text{Variables} & x & \in & \textit{Variables}
\end{array}
$$

The expressions contain entities, constants (other than entities), variables, the empty tuple (), n-ary tuples of expressions, and projection functions applied to expressions. The notation π^n_k stands for the kth projection function for expressions of arity n. The typing rules for expressions are as follows:

$$
\frac{E \in \textit{Entities}}{\Gamma \vdash E : \textit{entity}} \qquad \frac{K \in \textit{Constants}}{\Gamma \vdash K : \textit{const}}
$$

$$
\frac{}{\Gamma \vdash () : \textit{unit}} \qquad \frac{x \in \textit{Dom}(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1,...,e_n) : \tau_1 \times ... \times \tau_n} \qquad \frac{\Gamma \vdash e : \tau_1 \times ... \times \tau_n \qquad 1 \leq i \leq n}{\Gamma \vdash \pi^n_i(e) : \tau_i}
$$

The typing rules for constraints are also very simple:

$$
\frac{}{\Gamma \vdash \mathsf{true}} \qquad \frac{}{\Gamma \vdash \mathsf{false}}
$$

$$
\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = e_2} \qquad \frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1 \wedge c_2}
$$

The semantics for $\mathcal{C}_{tup}$ is identical to the semantics of $\mathcal{C}_{eq}$ except that equality between expressions is slightly more complex:

$\theta \models e_1 = e_2$ if $[\![\theta e_1]\!]$ and $[\![\theta e_2]\!]$ are equal where

$$\begin{array}{ll} [\![K]\!] & = K, \text{ if } K \in \mathit{Entities} \uplus \mathit{Constants} \\ [\![(e_1,...,e_n)]\!] & = ([\![e_1]\!],..,[\![e_n]\!]) \\ [\![\pi^n_k(e)]\!] & = \begin{cases} e_k, & \text{if } [\![e]\!] = (e_1,...,e_n) \\ \pi^n_k([\![e]\!]), & \text{otherwise} \end{cases} \end{array}$$

Any well-typed $\mathcal{C}_{tup}$ constraint c can be flattened to a constraint in which no expression has a product type: simplify all subconstraints $e_1 = e_2$ to $[\![e_1]\!] = [\![e_2]\!]$. Then, for some $\Gamma \vdash c$, repeatedly replace all subconstraints $e_1 = e_2$ where $\Gamma \vdash e_1 : \tau_1 \times ... \times \tau_n$ with

$$[\![\pi^n_1(e_1)]\!] = [\![\pi^n_1(e_2)]\!] \wedge ... \wedge [\![\pi^n_n(e_1)]\!] = [\![\pi^n_n(e_2)]\!]$$

until no product-typed expressions are left. For example, the constraint

$$(x,2) = ((3,y),y) \ \wedge \ z = \pi^2_1(z,y)$$

is rewritten as

$$\pi^2_1(x) = 3 \ \wedge \ \pi^2_2(x) = y \ \wedge \ 2 = y \ \wedge \ z = z.$$

Subsequently, such a rewritten constraint can be converted into an equivalent solved form, again by standard unification. However, here the unifier substitutes expressions for variables that may be prefixed by a number of tuple projection applications. For example, the solved form of the constraint from above would be $[3/\pi^2_1(x), 2/\pi^2_2(x), 2/y]$. The algorithms for satisfiability checking, subsumption checking and existential quantifier elimination for $\mathcal{C}_{eq}$ trivially extend to $\mathcal{C}_{tup}$.

Theorem 5.2.4. *$\mathcal{C}_{tup}$ is constraint compact.*

Proof. As in $\mathcal{C}_{eq}$, existential quantifier elimination does not introduce any new variables or constants. For the second condition, let C be a (possibly infinite) set of well-typed $\mathcal{C}_{tup}$-constraints such that only finitely many different variables and constants occur in the constraints. Let C' be the set of solved forms of the constraints of C. Converting these constraints to their solved forms does not introduce any new constants or variables, so there are still only finitely many of them in C'. Furthermore, as all variables can be given a finite type, the length of the tuple projection prefixes is bounded for each variable. Hence there can only be finitely many different solved forms, so C' is finite. Every constraint in C is equivalent to a solved form in C'. This implies constraint compactness. □

5.2.3 Inequalities and Arithmetic Constraints

In this section we define $\mathcal{C}_{\neq,<}$ by extending $\mathcal{C}_{tup}$ with inequalities between arbitrary expressions and gap-order constraints on integers. The syntax of $\mathcal{C}_{\neq,<}$ is given below.

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & \ldots \\
 & & \mid & N \\
\text{Constraints} & c & ::= & \ldots \\
 & & \mid & e_1 \neq e_2 \\
 & & \mid & e_1 <_g e_2, \qquad g \geq 0 \\
 & & \mid & c_1 \vee c_2 \\
\text{Integers} & N & \in & \mathbb{Z}
\end{array}
$$

A gap-order constraint is of the form $e_1 <_g e_2$ (where g is a positive integer) and is interpreted as $e_1 + g < e_2$. The reason $\mathcal{C}_{\neq,<}$ contains gap-order constraints rather than simple $<$-inequalities is that only gap-order constraints are closed under existential quantifier elimination. For example, consider $\exists y(x < y < z)$, which is equivalent to $x <_1 z$. We also introduce disjunctions ($\vee$) of constraints. The type system from $\mathcal{C}_{tup}$ is extended with new rules for typing expressions and disjunctions:

$$
\frac{N \in \mathbb{Z}}{\Gamma \vdash N : int} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \neq e_2}
$$

$$
\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 <_g e_2} \qquad \frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1 \vee c_2}
$$

The semantics of $\mathcal{C}_{\neq,<}$ is straightforward:

- $\theta \models e_1 \neq e_2$ if $[\![\theta e_1]\!]$ and $[\![\theta e_2]\!]$ are not unifiable.
- $\theta \models e_1 <_g e_2$ if θe_1 and θe_2 are ground and $[\![\theta e_1]\!] + g < [\![\theta e_2]\!]$.

However, solving $\mathcal{C}_{\neq,<}$-constraints is much harder than in the previous examples. For $\mathcal{C}_{eq}$ and $\mathcal{C}_{tup}$, the solved forms of constraints were simply substitutions. Here, a constraint is mapped to an equivalent *constraint graph*.

Constraint Graphs

The notion of a constraint graph is introduced to show that satisfiability checking, subsumption checking and existential quantifier elmination are computable and that $\mathcal{C}_{\neq,<}$ is constraint compact. The theory of constraint graphs is based on the gap graphs introduced by Revesz [Rev93]. Constraint graphs differ from gap graphs only in that the former may also contain inequalities between non-integers.

Definition 5.2.5. (**constraint graph**) *Let l and u be integer constants with $l < u$. An (l,u)-*constraint graph *(or simply* constraint graph, *if l and u are irrelevant) is a graph whose vertices are non-empty finite sets of $\mathcal{C}_{\neq,<}$-expressions. The edges are either undirected and labelled with $\neq$, or directed and labelled with $<_g$, for some integer $g \geq 0$. Moreover, a constraint graph satisfies the following conditions:*

1. *All vertices are pairwise mutually disjoint.*
2. *There is at most one edge between any pair of vertices (self-loops are allowed).*
3. *The expressions in the vertices are flat, i.e. they do not contain products.*

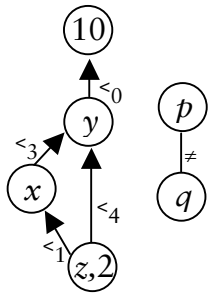

Figure 5.1: $(2,10)$-constraint graph representing the constraint $z = 2, z <_4 y, z <_1 x, x <_3 y, y <_0 10, p \neq q$.

4. *All expressions in a given vertex are of the same type.*
5. *The only integer constants occurring in the graph are l and u.*
6. *$\neq$-labelled (undirected) edges are only between vertices containing non-integer expressions.*
7. *$<_g$-labelled (directed) edges are only between vertices containing expressions of type int.*

Intuitively, the expressions inside a single vertex are all mutually related by equality constraints; if two vertices are connected by an undirected edge, their expressions are related by $\neq$; and if a vertex is connected to another one via a directed $<_g$-edge, the expressions of the first vertex are related to those in the second by $<_g$. Figure 5.1 shows a $(2,10)$-constraint graph representing the constraint $z = 2, z <_4 y, z <_1 x, x <_3 y, y <_0 10, p \neq q$.

Just as with constraints, we can define satisfiability for constraint graphs. The following definition also relates constraints graphs to constraints.

Definition 5.2.6. (satisfaction, equivalence) *A substitution θ* satisfies *a constraint graph G (we write $\theta \models G$) if:*

1. *For all non-singleton vertices $V = \{e_1, ..., e_n\}$, we have $\theta \models e_i = e_j$, $1 \leq i, j \leq n$.*
2. *For all $\neq$-labelled edges between vertices $V_1 = \{e_1, ...,\}$ and $V_2 = \{e'_1, ...,\}$, we have $\theta \models e_1 \neq e'_1$.*
3. *For all $<_g$-labelled edges from some vertex $V_1 = \{e_1, ...,\}$ to a vertex $V_2 = \{e'_1, ...,\}$, we have $\theta \models e_1 <_g e'_1$.*

A constraint graph G is satisfiable *if there is a substitution θ such that $\theta \models G$.*

A constraint graph G and a $C_{\neq,<}$-constraint c are equivalent *if $\theta \models c \iff \theta \models G$ for all substitutions θ.*

Satisfiability Checking

Instead of giving an algorithm for satisfiability checking of constraints, we specify one for constraint graphs. If we can then convert a constraint into an equivalent constraint graph (actually a set of constraint graphs, see Algorithm 5.2.13), the algorithm can also be used to check satisfiability of constraints. Lemma 5.2.9 proves the correctness of Algorithm 5.2.8.

Definition 5.2.7. (path, cycle, path length, root) *A* path *is a connected chain of directed edges between pairs of vertices* $(V_0, V_1), (V_1, V_2), ..., (V_{n-1}, V_n)$. *The path is a* cycle *if* V_0 *and* V_n *are identical. The* length *of a path is n, the number of edges in the path, plus the sum of all gap values in the labels of the edges belonging to the path. A vertex is a* root *if there is no directed edge pointing to it.*

Algorithm 5.2.8. (satisfiability) *Let G be an (l,u)-constraint graph. Let* V_l *and* V_u *be the vertices of G that contain l and u, respectively. Return true if all of the following conditions hold, and false otherwise.*

1. *G contains no cycle.*
2. *G contains no (undirected) self-loop.*
3. *No vertex contains two distinct constants.*
4. *There is no path from* V_u *to* V_l.
5. *If a longest path from* V_l *to* V_u *exists, its length is less than or equal to* $u - l$.

Lemma 5.2.9. *Given a constraint graph G, Algorithm 5.2.8 returns true if G is satisfiable, and false otherwise.*

Proof. Suppose the algorithm returns true. Then those connected components of the graph whose vertices are of non-integer types can be satisfied by assigning the same — but for each vertex a freshly chosen — constant value to the vertex, if the vertex contains only variables. Otherwise, if it does not contain only variables, it can be satisfied by assigning the value of the only constant in the vertex to all variables in the vertex. This assignment is possible and separates all non-integer vertices because we assume an infinite number of entities and constants, and all vertices are pairwise mutually disjoint.

Now for the subgraph with vertices of integer type, we first assign l to V_l and u to V_u. All other vertices contain only variables. Let $lp(V, V')$ denote the length of the longest path from a vertex V to a vertex V' in G. For each vertex V with a path leading to it from V_l or V_u, assign $max(l + lp(V_l, V), u + lp(V_u, V))$. This assignment satisfies all outgoing edges of vertices reachable from V_l or V_u.

Let m be the length of the longest path in G. Assign the value $l - m$ to each root vertex (if it is not V_l or V_u). Now the only unassigned vertices are non-root vertices that are not reachable from neither l nor u. Assign to these vertices V the maximum of the values $(l - m) + max_X(lp(X, V))$ where X is any root vertex in G. This assignment satisfies all outgoing edges of vertices reachable from a root vertex but not from either V_l nor V_u.

Suppose the algorithm returns false. Then G either contains a cycle or there is a path from V_u to V_l or the length of the longest path from V_l to V_u is greater than $u - l$. In all these cases G is clearly not satisfiable. □

Conjunction

The algorithm for converting constraints into constraint graphs works by first converting all atomic constraints of a conjunct into a constraint graph and then combining them. We therefore first need an algorithm for combining two constraint graphs. Algorithm 5.2.11 is proven correct by Lemma 5.2.12.

Definition 5.2.10. (constraint graph conjunction) *Let* G_1 *and* G_2 *be two constraint graphs.* G *is a conjunction of* G_1 *and* G_2 *if for all substitutions* θ

$$\theta \models G \iff (\theta \models G_1 \text{ and } \theta \models G_2).$$

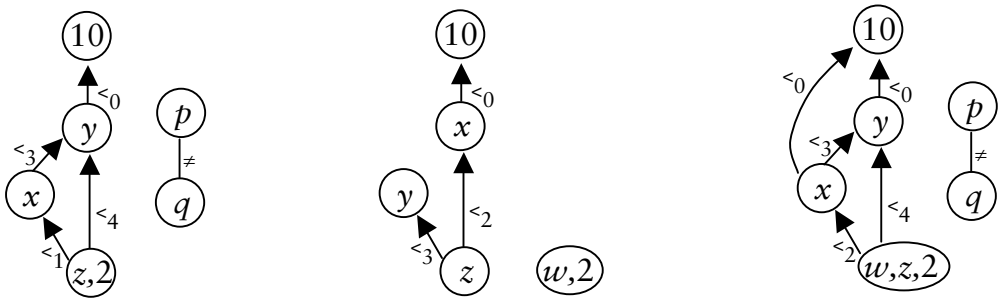

Figure 5.2: The constraint graph on the right panel is the conjunction of the two constraint graphs on the left.

Algorithm 5.2.11. (constraint graph conjunction) *Let $G_1 = (\mathcal{V}_1, \mathcal{E}_1)$ and $G_2 = (\mathcal{V}_2, \mathcal{E}_2)$ be two (l,u)-constraint graphs. Let $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$ and $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$. Repeatedly replace pairs of vertices in $\mathcal{V}$ that are not disjoint by their union, until all vertices are pairwise mutually disjoint. Call this new set of vertices $\mathcal{V}''$. Let*

$$\begin{aligned}\mathcal{E}' = \quad & \{(V_1' \overset{\neq}{\text{---}} V_2') \mid \quad V_1', V_2' \in \mathcal{V}'', \\ & \qquad\qquad V_1, V_2 \in \mathcal{V}, \\ & \qquad\qquad V_1 \subseteq V_1', V_2 \subseteq V_2', \\ & \qquad\qquad (V_1 \overset{\neq}{\text{---}} V_2) \in \mathcal{E}\} \cup \\ & \{(V_1' \xrightarrow{<_g} V_2') \mid \quad V_1', V_2' \in \mathcal{V}'', \\ & \qquad\qquad V_1, V_2 \in \mathcal{V}, \\ & \qquad\qquad V_1 \subseteq V_1', V_2 \subseteq V_2', \\ & \qquad\qquad (V_1 \xrightarrow{<_g} V_2) \in \mathcal{E}\}\end{aligned}$$

If there are self-loops (directed or undirected) or there are two opposite directed edges $V_1 \xrightarrow{<_g} V_2$ and $V_2 \xrightarrow{<_{g'}} V_1$ in $\mathcal{E}'$ then simply return any unsatisfiable (l,u)-constraint graph, for example $\{u\} \xrightarrow{<_0} \{l\}$. Otherwise, if there are multiple edges between any pair of vertices in $\mathcal{E}'$, remove all but the one labelled with the largest gap-value. Call this new set of edges $\mathcal{E}''$.

Return the (l,u)-constraint graph $G' = (\mathcal{V}'', \mathcal{E}'')$.

Figure 5.2 shows the result of the conjunction of two constraints graphs. The following lemma proves the correctness of the conjunction algorithm.

Lemma 5.2.12. *Given two (l,u)-constraint graphs G_1 and G_2, Algorithm 5.2.11 returns a new (l,u)-constraint graph G that is the conjunction of both graphs.*

Proof. It is easy to see that G is indeed a (l,u)-constraint graph. We have to show that for all substitutions θ, $\theta \models G$ iff $\theta \models G_1$ and $\theta \models G_2$.

First assume $\theta \models G$. Consider any non-singleton vertex $V = \{e_1, ..., e_n\}$ in G_1 or in G_2. Then, by construction of G, there is a vertex V' in G such that $V \subseteq V'$. Since θ satisfies G, it equalises all expressions within V' and hence also within V, by definition of constraint graph satisfaction. Now consider any edge in G_1 or in G_2 of the form $V_1 \overset{\neq}{\text{---}} V_2$. Then by construction of G, there exists an edge $V_1' \overset{\neq}{\text{---}} V_2'$ in G such that $V_1 \subseteq V_1'$ and $V_2 \subseteq V_2'$.

Therefore, given any $e_1 \in V_1, e_2 \in V_2$ we have $\theta \models e_1 \neq e_2$, as required. Finally, consider any edge in G_1 or in G_2 of the form $V_1 \xrightarrow{<_g} V_2$. Then by construction of G, there exists an edge $V_1' \xrightarrow{<_{g'}} V_2'$ such that $V_1 \subseteq V_1'$, $V_2 \subseteq V_2'$ and $g' \geq g$. Therefore, given any $e_1 \in V_1, e_2 \in V_2$ we have $\theta \models e_1 <_{g'} e_2$ and hence also $\theta \models e_1 <_g e_2$, as required. Hence θ also satisfies both G_1 and G_2.

For the other direction, assume $\theta \models G_1$ and $\theta \models G_2$. Consider any two expressions e_i and e_j within a non-singleton vertex $V = \{e_1, ..., e_n\}$ in G. Then either there is a vertex V' in G_1 or in G_2 with $e_i, e_j \in V'$, in which case clearly $\theta \models e_i = e_j$; or, by construction of G, there is a chain of vertices $V_1, V_2, ..., V_n$ drawn alternately from G_1 and G_2 such that $e_i \in V_1$, $e_j \in V_n$ and $V_1 \cap V_2 \neq \emptyset$, ..., $V_{n-1} \cap V_n \neq \emptyset$. In this case θ equalises all members of the chain of vertices, in particular also e_i and e_j, as required. Now consider any edge in G of the form $V_1' \xrightarrow{\neq} V_2'$ and $e_1 \in V_1'$ and $e_2 \in V_2'$. Then there must be an edge $V_1 \xrightarrow{\neq} V_2$ in G_1 or in G_2 such that V_1 and V_2 are subsets of V_1' and V_2', respectively. Since θ makes all expressions in V_1 unequal to those in V_2 and since θ also equalises all expressions within V_1' and within V_2', respectively, θ also makes e_1 and e_2 unequal, as required. In a similar manner it can be shown that any edge in G of the form $V_1' \xrightarrow{<_g} V_2'$ is also satisfied by θ. Therefore, θ also satisfies G. □

Constraint Conversion

We now have all the tools needed to convert a constraint into a set of constraint graphs. The algorithm essentially converts the constraint into DNF and then, using Algorithm 5.2.11, converts each conjunct into a constraint graph. Lemma 5.2.15 proves the correctness of the Algorithm 5.2.13.

Algorithm 5.2.13. (constraint to graph conversion) *Let c be a well-typed $\mathcal{C}_{\neq,<}$-constraint, and $\Gamma \vdash c$. Let l and u be two integers such that l is less than the least integer constant in c and u is greater than or equal to the largest integer constant in c. If c does not have any integer constant, let l and u be any integers satisfying $l < u$. The following algorithm converts c into an equivalent (l,u)-constraint graph.*

1. *To flatten all expressions, simplify all subexpressions e to $[\![e]\!]$. Then repeatedly replace all subconstraints of the form $e_1 = e_2$ where $\Gamma \vdash e_1 : \tau_1 \times ... \times \tau_n$ with*

   $$[\![\pi_1^n(e_1)]\!] = [\![\pi_1^n(e_2)]\!] \wedge ... \wedge [\![\pi_n^n(e_1)]\!] = [\![\pi_n^n(e_2)]\!],$$

   and subconstraints of the form $e_1 \neq e_2$ where $\Gamma \vdash e_1 : \tau_1 \times ... \times \tau_n$ with

   $$[\![\pi_1^n(e_1)]\!] \neq [\![\pi_1^n(e_2)]\!] \vee ... \vee [\![\pi_n^n(e_1)]\!] \neq [\![\pi_n^n(e_2)]\!]$$

   until no product-typed expressions are left.
2. *Replace subconstraints of the form $e_1 \neq e_2$ where $\Gamma \vdash e_1 : int$ by*

   $$e_1 <_0 e_2 \vee e_2 <_0 e_1.$$

3. *Replace subconstraints of the form $e = N$ where the integer constant N satisfies $l < N < u$ by $e <_0 (N+1) \wedge (N-1) <_0 e$. At this point, all subconstraints are equalities or inequalities between flat non-integer expressions, gap-order constraints, or equality constraints between integer expressions.*

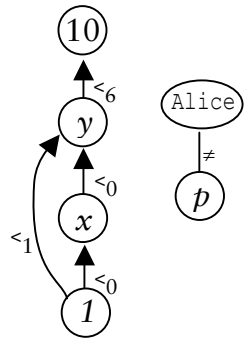

Figure 5.3: Conversion of the constraint $1 < x, x < y, y = 3, p \neq \texttt{Alice}$ into a $(1,10)$-constraint graph.

4. *Convert to DNF, and perform trivial simplifications, thereby eliminating all occurrences of* $\top$, $\bot$, *constraints between constants (e.g.* $3 = 2$*) and inequalities between the same expressions (e.g.* $x_{int} >_0 x_{int}$*).*
5. *Compute a constraint graph G for each disjunct by first converting the atomic constraints into* (l,u)*-constraint graphs and then combining them with the constraint graph conjunction algorithm. Atomic constraints are converted as follows:*

   $e = l$ *is converted into* $\{e,l\} \xrightarrow{<_{u-l-1}} \{u\}$.

   $e = u$ *is converted into* $\{l\} \xrightarrow{<_{u-l-1}} \{e,u\}$.

   Otherwise, $e = e'$ *is converted into* $\{e,e'\}$.

   $e \neq e'$ *is converted into* $\{e\} \xrightarrow{\neq} \{e'\}$.

   $e <_g e'$*, where* e *and* e' *are not constants, is converted into* $\{e\} \xrightarrow{<_g} \{e'\}$.

   $e <_g N$*, where* N *is an integer constant, is converted into* $\{e\} \xrightarrow{<_{g+u-N}} \{u\}$.

   Similarly, $N <_g e$ *is converted into* $\{l\} \xrightarrow{<_{g+N-l}} \{e\}$.
6. *Return the (disjunctive) set of constraint graphs of the disjuncts.*

Figure 5.3 shows the result of converting the constraint $1 < x, x < y, y = 3, p \neq \texttt{Alice}$ into a $(1,10)$-constraint graph. We will now prove the correctness of the constraint graph conversion algorithm.

Definition 5.2.14. *A* $C_{\neq,<}$*-constraint c is* equivalent *to a finite set of constraint graphs* $\vec{G}$ *if, for all substitutions* θ*,*

$$\theta \models c \iff \theta \models G_i \text{ for some } G_i \in \vec{G}.$$

Lemma 5.2.15. *Let c be a* $C_{\neq,<}$*-constraint. Algorithm 5.2.13 converts c into an equivalent set of* (l,u)*-constraint graphs* $\{G_1, ..., G_n\}$*, for some integers l and u.*

Proof. The first four steps are all simple transformations that do not change the meaning of the constraint. Step 5 is correct because Algorithm 5.2.11 is correct, and the conversion rules for atomic constraints are correct and exhaustive.

For example, consider the atomic constraint $c' \equiv e <_g N$. Assume θ satisfies c'. Then the edge between e and u is satisfied since $e <_{g+u-N} u$ is equivalent to $e <_g N$. The other direction is similar: assuming that θ satisfies the constraint graph $\{e\} \xrightarrow{<_{g+u-N}} \{u\}$, we must have $\theta \models e <_{g+u-N} u$ and thus $\theta \models c'$. □

Subsumption Checking

As in the case of satisfiability checking, we provide an algorithm for checking subsumption of constraint graphs rather than constraints directly. Algorithm 5.2.20 rests on the notions of an *underlying graph*, *gap tuples* and *independent* sets (Definitions 5.2.16, 5.2.17, and 5.2.18). These concepts will also be needed for existential quantifier elimination and the proof of constraint compactness. The correctness of Algorithm 5.2.20 is given by Lemma 5.2.21.

Definition 5.2.16. *Let G be a constraint graph. Then the* underlying graph *of G is the graph obtained by erasing the labels on the edges of G.*

Definition 5.2.17. *Let G be a constraint graph and σ an ordering of the directed edges of G. The* σ-gap tuple *of G is the tuple of the gap values of the edges of G in σ-order.*

Definition 5.2.18. *Let $P = (p_1, ..., p_k)$ and $Q = (q_1, ..., q_k)$ be two k-tuples of natural numbers ($k \geq 0$). We define a well-founded partial order $\leq$ on k-tuples such that $P \leq Q$ if $p_1 \leq q_1 \wedge ... \wedge p_k \leq q_k$. We say P and Q are* independent *if $P \not\leq Q$ and $Q \not\leq P$. A set of k-tuples of natural numbers is* independent *if all members of the set are mutually pairwise independent.*

Definition 5.2.19. *Let G_1 and G_2 be two (l,u)-constraint graphs. G_1 is* subsumed *by G_2 if, for all substitutions θ,*

$$\theta \models G_1 \text{ implies } \theta \models G_2.$$

Algorithm 5.2.20. *Given two (l,u)-constraint graphs G_1 and G_2, this algorithm returns true only if G_1 is subsumed by G_2 (the converse does not hold in general).*
Return true if

- *G_1 and G_2 have the same underlying graph $\hat{G}$ and*
- *given an ordering σ on the directed edges of $\hat{G}$, $\vec{g}_2 \leq \vec{g}_1$ where $\vec{g}_1$ and $\vec{g}_2$ are the σ-gap tuples of G_1 and G_2, respectively.*

Return false, otherwise.

Figure 5.4 depicts a constraint graph of $z = 2, z <_0 y, z <_1 x, x <_1 y, y <_0 10, p \neq q$ that subsumes the one from Figure 5.1. Note that both graphs have the same underlying graph, but the $<$-labels on this one are less than or equal to those from Figure 5.1. The following lemma proves that Algorithm 5.2.20 computes a correct approximation of the subsumption relation on constraint graphs.

Lemma 5.2.21. *Let G_1 and G_2 be two (l,u)-constraint graphs. If Algorithm 5.2.20 returns true then G_1 is subsumed by G_2.*

Proof. Assume θ satisfies G_1. Since G_1 and G_2 have the same underlying graph, θ satisfies G_2's equality and inequality constraints. Any directed edge $V \xrightarrow{<_{g_2}} V'$ in G_2 is satisfied by θ since θ also satisfies $V \xrightarrow{<_{g_1}} V'$ in G_1 where $g_1 \geq g_2$.

Therefore θ also satisfies G_2. □

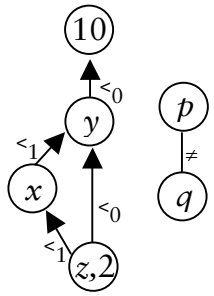

Figure 5.4: The constraint graph from Figure 5.1 is subsumed by this $(2,10)$-graph, according to Algorithm 5.2.20.

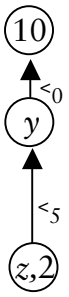

Figure 5.5: Existentially eliminating the variables x and p from the constraint graph from Figure 5.1 yields $z = 2, z <_5 y, y <_0 10$.

Existential Quantifier Elimination

The algorithm for existential quantifier elimination on constraint graphs completes the set of operations that are required for constraint domains. This is summarised in Theorem 5.2.24.

Algorithm 5.2.22. *To perform existential quantifier elimination of a variable x from a constraint graph G, remove from all vertices those expressions in which x occurs. If the resulting vertex V is empty, replace all pairs of edges $V_1 \xrightarrow{<_{g_1}} V \xrightarrow{<_{g_2}} V_2$ by $V_1 \xrightarrow{<_g} V_2$ where $g = g_1 + g_2 + 1$.*

Remove V with all incident edges from the graph. If between any pair of vertices there is more than one directed edge pointing in the same direction, delete all but the one with the largest gap value from G. If between any pair of vertices there are directed edges pointing in opposite directions, return the unsatisfiable constraint graph $\{u\} \xrightarrow{<_0} \{l\}$.

Figure 5.5 shows the constraint graph obtained by eliminating the variables x and p from the constraint graph from Figure 5.1. Note that q is also removed in the process, and that x is bypassed by a $<_5$-edge. The following lemma proves the correctness of Algorithm 5.2.22.

Lemma 5.2.23. *Let G be a constraint graph. Then the constraint graph G' obtained by*

eliminating the variable x from G using Algorithm 5.2.22 satisfies

$$\theta \models G' \iff \textit{for some expression } e,\ \theta[e/x] \models G$$

Proof. For simplicity's sake, we assume that x is not of a product type. The proof easily generalises to the case where x is a product.

First assume $\theta \models G'$. If in the elimination process x is removed from a vertex V in G with at least two elements, x and some other expression e, say, then we have $\theta[e/x] \models G$. If the vertex V in G from which x is removed contains only x and has $\neq$-edges connecting it to vertices $V_1, ..., V_n$ then K can be any constant different from θe_1,...,θe_n, where e_i is any expression in V_i. In the other case, if V contains only x and has incident gap order edges, let $V_1, ..., V_p$ and $g_1, ..., g_p$ be the vertices with an outgoing directed edge to V, and the gap values on these edges, respectively, and let $W_1, ..., W_q$ and $h_1, ..., h_q$ be the vertices with an incoming directed edge from V, and the gap values on these edges, respectively. Choose s and t such that the value of an element in V_s under θ plus g_s is maximal (and call it M) and the value of an element in W_t under θ minus h_t is minimal (call it m). If we can choose K such that $m < K < M$, then $\theta[K/x]$ clearly satisfies all edges incident to V, and thus the entire graph G. The algorithm adds the edge $V_s \xrightarrow{g_s+h_t+1} V_t$ to produce G', and θ satisfies it by assumption. This implies that $m+1 < M$, so there is indeed a K between m and M.

For the other direction, assume $\theta[e/x] \models G$, for some e. It is sufficient to show that θ satisfies the new directed edges in G'. If G' contains a new edge $V_1 \xrightarrow{<_g} V_2$ then G must contain $V_1 \xrightarrow{<_{g_1}} \{x\} \xrightarrow{<_{g_2}} V_2$ where $g = g_1 + g_2 + 1$. Let e_1 and e_2 be expressions in V_1 and V_2, respectively. By assumption, $\theta e_1 + g_1 < \theta e$ and $\theta e + g_2 < \theta e_2$. This implies $\theta e_1 <_g \theta e_2$, as required. □

Theorem 5.2.24. *Satisfiability checking, subsumption checking and existential quantifier elimination for $\mathcal{C}_{\neq,<}$-constraints are computable.*

Proof. This follows directly from the soundness of converting constraints into constraint graphs (Lemma 5.2.15) and from the soundness of the constraint graph algorithms for satisfiability checking (Lemma 5.2.9), for subsumption checking (Lemma 5.2.21) and for existential quantifier elimination (Lemma 5.2.23). □

Constraint Compactness

To prove constraint compactness of $\mathcal{C}_{\neq,<}$, we first have to prove a few auxiliary lemmas. Lemma 5.2.25 is due to Revesz [Rev93]. Lemma 5.2.28 proves compactness for single constraint graphs, and Lemma 5.2.29 generalises that result to disjunctions of constraint graphs. Finally, Theorem 5.2.30 proves constraint compactness of $\mathcal{C}_{\neq,<}$.

Lemma 5.2.25. *Every independent set of k-tuples of natural numbers is finite ($k \geq 0$).*

Proof. We prove the statement by induction on k. For $k = 0$, the statement is trivial.

Let I be a non-empty independent set of $(k+1)$-tuples with $A = (a_1, ..., a_{k+1}) \in I$. Given any other tuple $P = (p_1, ..., p_{k+1}) \in I$, there must be some $i \in \{1, ..., k+1\}$ such that $p_i < a_i$, since A and P are independent. Therefore, every tuple in I other than A belongs to at least one of the sets $S_{i,p} = \{(p_1, ..., p_{k+1}) \in I \mid p_i = p\}$, where $i \in \{1, ..., k+1\}$ and $0 \leq p < a_i$.

There are only finitely many of these sets, so it is sufficient to prove that any $S_{i,p}$ is finite. $S_{i,p}$ is also an independent set as it is a subset of I. Let $S'_{i,p}$ be the set of k-tuples constructed

from the $(k+1)$-tuples in $S_{i,p}$ without their ith component. Then $S'_{i,p}$ is an independent set of k-tuples, for the ith components of the $(k+1)$-tuples in $S_{i,p}$ are all equal. By the induction hypothesis, $S'_{i,p}$ is finite, and hence $S_{i,p}$ is finite as well. □

Definition 5.2.26. *Let S be a set of k-tuples of natural numbers ($k \geq 0$). A set C is a* cover *of S (C covers S) if for all tuples P in S there is a tuple Q in C such that $Q \leq P$.*

Lemma 5.2.27. *Let S be a set of k-tuples of natural numbers ($k \geq 0$). Then there is a finite independent set $C \subseteq S$ that covers S.*

Proof. Let $C^- = \{P \in S \mid \exists Q \in S.\ P \neq Q \text{ and } Q \leq P\}$ be the set of all tuples in S for which there is a smaller tuple in S. Let $C = S - C^-$, so $S = C \uplus C^-$. S is clearly a cover of itself, therefore for all tuples P in S there is a tuple Q either in C or in C^- such that $Q \leq P$. In the latter case, by well-foundedness of $(S, \geq)$ and by construction of C^-, there must be a $Q_0 \in S$ such that $Q_0 \leq Q$ and Q_0 is a minimal element in S. Since Q_0 is minimal, it cannot be in C^-, thus $Q_0 \in C$. By transitivity of $\leq$, $Q_0 \leq P$. Hence C is also a cover of S.

Moreover, C is an independent set by construction, and by Lemma 5.2.25, C is finite. □

Lemma 5.2.28. *Let $\mathcal{G}$ be a set of (l,u)-constraint graphs such that only finitely many different variables and constants occur in the vertices of the graphs. Then there exists a finite subset $\mathcal{G}_{fin} \subseteq \mathcal{G}$ such that for every graph $G \in \mathcal{G}$ there is a graph $G' \in \mathcal{G}_{fin}$ such that $G \Rightarrow G'$.*

Proof. Since there are only finitely many different variables and constants, there can only be finitely many distinct vertices. Moreover, since graphs can have at most one edge between any pair of vertices, there are only a finite number of distinct underlying graphs of constraint graphs in $\mathcal{G}$. Therefore, it is sufficient to show that for any of these finitely many distinct underlying graphs $\hat{G}$ there exists a finite set $\mathcal{G}_{fin} \subseteq \mathcal{G}$ that covers all those graphs in $\mathcal{G}$ with underlying graph $\hat{G}$.

Choose any ordering σ on $\hat{G}$'s directed edges. Let S be the set of the σ-gap tuples of those graphs in $\mathcal{G}$ whose underlying graph is $\hat{G}$. By Lemma 5.2.27, there is a finite subset S_{fin} of S that covers S. Choose $\mathcal{G}_{fin} \subseteq \mathcal{G}$ to be the finite set of constraint graphs with σ-gap tuples in S_{fin}. □

Lemma 5.2.29. *Let $\vec{\mathcal{G}}$ be a set of finite disjunctions of (l,u)-constraint graphs such that only finitely many different variables and constants occur in the vertices of the graphs. Then there exists a finite subset $\vec{\mathcal{G}}_{fin} \subseteq \vec{\mathcal{G}}$ such that for every disjunction of graphs $\mathcal{G} \in \vec{\mathcal{G}}$ there is a disjunction of graphs $\mathcal{G}' \in \vec{\mathcal{G}}_{fin}$ such that $\mathcal{G} \Rightarrow \mathcal{G}'$, i.e.*

$$\forall G \in \mathcal{G}.\ \exists G' \in \mathcal{G}'.\ G \Rightarrow G'.$$

Proof. By Lemma 5.2.28, there is a positive integer N such that, for all disjunctions in $\vec{\mathcal{G}}$ with size greater than N, the disjunction is equivalent to one with size at most N and can thus be ignored. Again by Lemma 5.2.28, for all n with $2 \leq n \leq N$, there are only finitely many constraint graph disjunctions of size n whose graphs are all pairwise mutually not related by $\Rightarrow$, all others are therefore equivalent to a disjunction of size strictly less than n. Therefore, all but a finite number of disjunctions in $\vec{\mathcal{G}}$ are equivalent to a single constraint graph (as opposed to a disjunction). By Lemma 5.2.28, there is a finite subset $\vec{\mathcal{G}}'_{fin}$ of $\vec{\mathcal{G}}$ that covers the set of these constraint graphs. Then we can choose $\vec{\mathcal{G}}_{fin}$ to be the union of the finitely many remaining disjunctions and $\vec{\mathcal{G}}'_{fin}$. □

Interestingly, the proof for Lemma 5.2.29 does not rely on any properties of constraint graphs other than the compactness property stated in Lemma 5.2.28. It can therefore be generalised to other constraint domains: adding disjunction to a constraint-compact constraint domain preserves constraint compactness.

Theorem 5.2.30. (constraint compactness) *$\mathcal{C}_{\neq,<}$ is constraint compact.*

Proof. First of all, existential quantifier elimination does not introduce any new variables or constants, by construction of Algorithm 5.2.22. For the second condition, any set of constraints can be converted into an equivalent set of disjunctions of (l,u)-constraint graphs where l and u are a lower bound and an upper bound, respectively, of the integer constants occurring in the set. Since the conversion does not introduce any new variables and constants, constraint compactness follows from the soundness of the conversion algorithm (Lemma 5.2.15) and from the compactness of constraint graphs (Lemma 5.2.29). □

5.2.4 Sets and Built-In Functions

The design of the constraint domain $\mathcal{C}_0$ was guided by the EHR case study. In $\mathcal{C}_0$, we add to $\mathcal{C}_{\neq,<}$ set expressions and constraints, and built-in functions. The function symbols are interpreted by a fixed set of side-effect-free[1] functions with finite domain that may return environment-dependent data. For example, for our case study we have functions to access data fields of health record items such as `Get-EHR-item-author`(pat,id), and a function `Current-time`() that returns the current time. The syntax is as follows:

Expressions	e	::=	...
		|	$F(e_1,...,e_n)$
		|	$\emptyset$
		|	Ω_τ
		|	$\{e_1,...,e_n\}$
		|	$e_1 - e_2$
		|	$e_1 \cap e_2$
		|	$e_1 \cup e_2$
Constraints	c	::=	...
		|	$e_1 \subseteq e_2$
Functions	F	$\in$	*FunctionNames*

The constant expression Ω_τ represents the universal set for a specific type. In combination with the other set constructs, we can express finite and cofinite sets. We usually omit the subscript τ if it is clear from the context. From the given constraint constructs, various other useful ones can be derived, for example

- $e \in [e_1,e_2]$ stands for $e \leq e_1 \wedge e \leq e_2$;
- $[e_1,e_2] \subseteq [e'_1,e'_2]$ for $e'_1 \leq e_1 \wedge e_2 \leq e'_2$;
- $e_1 \in e_2$ for $\{e_1\} \subseteq e_2$;
- and $e_1 \notin e_2$ for $\{e_1\} \subseteq \Omega - e_2$.

[1] This is not a strict requirement but policy writers should not rely on how often or in which order functions are called during evaluation.

Every function symbol F is associated with types for its domain and its codomain, denoted by $Dom(F)$ and $Cod(F)$, respectively. The type system for $\mathcal{C}_0$ contains all rules from the type system for $\mathcal{C}_{\neq,<}$, as well as the following ones:

$$\frac{\Gamma \vdash e : \tau_1 \qquad F \in \mathit{FunctionNames} \qquad Dom(F) = \tau_1 \qquad Cod(F) = \tau_2}{\Gamma \vdash F(e) : \tau_2}$$

$$\frac{}{\Gamma \vdash \emptyset : set(\tau)} \qquad \frac{}{\Gamma \vdash \Omega_\tau : set(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \cdots \quad \Gamma \vdash e_n : \tau}{\Gamma \vdash \{e_1, ..., e_n\} : set(\tau)} \qquad \frac{\Gamma \vdash e_1 : set(\tau) \qquad \Gamma \vdash e_2 : set(\tau)}{\Gamma \vdash e_1 - e_2 : set(\tau)}$$

and similarly for $\cap$ and $\cup$.

$$\frac{\Gamma \vdash e_1 : set(\tau) \qquad \Gamma \vdash e_2 : set(\tau)}{\Gamma \vdash e_1 \subseteq e_2}$$

Each function symbol F is interpreted by a function $[\![F]\!]$ with a finite domain $Dom(F)$ and codomain $Cod(F)$, so

$$[\![F(e)]\!] = [\![F]\!]([\![e]\!]).$$

The condition that the function be finite is essential for constraint compactness: it is easy to see that compactness, and thus decidability, is broken by allowing a successor function.

Set expressions and subset constraints are interpreted as expected:

$$\begin{array}{lcl}
[\![\emptyset]\!] & = & \emptyset \\
[\![\Omega_\tau]\!] & = & \Omega_\tau \\
[\![\{e_1, ..., e_n\}]\!] & = & \{[\![e_1]\!], ..., [\![e_n]\!]\} \\
[\![e_1 - e_2]\!] & = & [\![e_1]\!] - [\![e_2]\!] \\
[\![e_1 \cap e_2]\!] & = & [\![e_1]\!] \cap [\![e_2]\!] \\
[\![e_1 \cup e_2]\!] & = & [\![e_1]\!] \cup [\![e_2]\!]
\end{array}$$

$\theta \models e_1 \subseteq e_2$ if $[\![\theta e_1]\!] \subseteq [\![\theta e_2]\!]$

Finite functions can be represented as finite sets, so they can be ignored in the analysis of the constraint compactness of $\mathcal{C}_0$. However, solving set constraints is much harder (see [PP97, Koz94, Aik94] for an overview of various approaches). Instead of presenting algorithms to solve the general form of set constraints, we use groundness analysis on the policy rules to ensure that, whenever existential quantifier elimination is to be performed on a $\mathcal{C}_0$-constraint, all set expressions are ground. All set sub-constraints occurring within a constraint then trivially reduce to true or false, and in particular do not introduce any new constants when existential quantifier elimination is performed. The resulting constraint is always from $\mathcal{C}_{\neq,<}$ which we already know to be constraint compact. While this argument does not establish the constraint compactness of $\mathcal{C}_0$, it does prove termination of the evaluation algorithm presented in §6.2 for $\mathcal{C}_0$ policies, under the groundness assumption. Groundness analysis of CASSANDRA policies is discussed in detail in §6.3.

6 Query Evaluation

In policy-based trust management systems, access control decisions are made through queries to the local policy. Query evaluation algorithms for deductive logic databases can be categorised into bottom-up and top-down algorithms. Bottom-up algorithms are closely based on the fixed point semantics: starting from basic facts, new derived facts are iteratively added until a fixed point is reached; the model can thus be pre-computed and reused. Top-down algorithms, on the other hand, are based on some form of resolution and are goal-oriented, i.e., if the query is partially instantiated (which is normally the case in CASSANDRA) this information can be used to prune the search space for efficiency. The disadvantages are that answers are not pre-computed, and standard top-down query evaluation algorithms such as Prolog-style SLD resolution are often not termination-complete [CW93, CW96].

For most deductive database applications, the bottom-up approach is the preferred choice as the model has to be computed only once (as long as the database is not modified) and can then be efficiently used to compute answers. Moreover, the danger of non-termination of query evaluation is usually highly undesirable.

However, bottom-up evaluation is not suitable for CASSANDRA for several reasons:

- The constraint domain (e.g. $\mathcal{C}_0$) may contain function calls that depend on the environment, for example for getting the current time, and therefore cannot be pre-computed.
- The fact that rule bodies can refer to remote predicates would require a distributed form of bottom-up evaluation which would be impractical.
- Requests made to the access control engine can modify policies. For example, a successful role activation request adds a hasActivated credential rule to the policy. The model would thus have to be re-computed after every such request.

SLG resolution (Linear resolution with Selection function for General logic programs) is an algorithm due to Chen and Warren [CW93, CW96] that is sound and complete with re-

spect to the well-founded semantics for logic programs with negation. Toman [Tom97] extended the algorithm for positive Datalog$^{\mathcal{C}}$ and proved that query evaluation for constraint-compact constraint domains $\mathcal{C}$ is guaranteed to terminate whenever the bottom-up approach terminates. Moreover, the complexity of the extended resolution algorithm, SLG$^{\mathcal{C}}$, is no worse than the complexity of bottom-up algorithms; in practice, it is significantly faster [Tom95].

§6.1 reviews SLG$^{\mathcal{C}}$ resolution in detail. §6.2 extends SLG$^{\mathcal{C}}$ resolution to CASSANDRA's policy language. In §6.3, we discuss the uses of *groundness analysis* and show that the SLG$^{\mathcal{C}}$ algorithm can be used to perform groundness analysis on policies, when the constraints are mapped to an abstract domain. Less mathematically-inclined readers may wish to skip §6.3.

6.1 SLG$^{\mathcal{C}}$ Resolution

SLG$^{\mathcal{C}}$ resolution is a top-down, goal-oriented evaluation algorithm for Datalog$^{\mathcal{C}}$ programs. It preserves the termination and complexity properties of bottom-up algorithms. With partially or fully instantiated queries, evaluation is usually significantly more efficient. The strong termination property is achieved by a *memoing* or *tabling* strategy. Answers to a goal are stored in a table indexed by goals, so to solve a goal for which a suitable table entry already exists, the algorithm uses the tabled answers as solutions. Whenever new answers are added to the table, they are automatically propagated to other waiting evaluation branches. Only if no relevant entry exists for the goal, a proof tree together with a new table entry is created and populated.

The algorithm and results are due to Toman [Tom97].

Algorithm 6.1.1. (SLG$^{\mathcal{C}}$ resolution) *Let $P_0,...,P_k$ be predicates, c,c' constraints from a constraint domain $\mathcal{C}$, and $\mathcal{P}$ be a Datalog$^{\mathcal{C}}$ program. An* SLG$^{\mathcal{C}}$ tree *consists of nodes of the form*

- $\mathsf{root}(P_0;c)$,
- $\mathsf{body}(P_0;[P_1,...,P_k];c)$,
- $\mathsf{goal}(P_0;(P_1,c');[P_2,...,P_k];c)$, *and*
- $\mathsf{ans}(P;c)$.

An SLG$^{\mathcal{C}}$ proof forest for the query $(P_0 \leftarrow c_0)$ is built by expanding an SLG$^{\mathcal{C}}$ tree with an initial node $\mathsf{root}(P_0;c_0)$ with the following rules as long as they can be applied. Let $\mathsf{Ans}(P,c)$ be the set of all constraints c' such that $\mathsf{ans}(P,c')$ is in an SLG$^{\mathcal{C}}$ tree with $\mathsf{root}(P,c)$, and c' is not subsumed by any older answer in that tree.

- ***Clause Resolution:*** *A node $\mathsf{root}(P;c)$ has child nodes of the form*

  $$\mathsf{body}(P;[P_1,...,P_k];c \wedge d)$$

  for all $P_1,...,P_k,d$ such that $P \leftarrow P_1,...,P_k,d \in \mathcal{P}$ and $c \wedge d$ is satisfiable.
- ***Query Projection:*** *A node $\mathsf{body}(P;[P_1,...,P_k];c)$ has child nodes of the form*

  $$\mathsf{goal}(P;(P_1,c');[P_2,...,P_k];c)$$

  for all[1] $c' \in \exists_{-P_1}(c)$.

[1] $\exists_{-P}(c)$ is shorthand for $\exists_{-Fv(P)}(c)$.

- ***Answer Propagation:*** *A node* $\mathsf{goal}(P;(P_1,c');[P_2,...,P_k];c)$ *has child nodes of the form*

  $$\mathsf{body}(P;[P_2,...,P_k];c\wedge a)$$

  for all $a\in \mathit{Ans}(P_1,c'')$ *where* $c'\Rightarrow c''$ *and* $c\wedge a$ *is satisfiable.*
- ***Answer Projection:*** *A node* $\mathsf{body}(P;[];c)$ *has child nodes of the form*

  $$\mathsf{ans}(P;a)$$

  for all $a\in\exists_{-P}(c)$.

Then we have the following important results:

Theorem 6.1.2. *Algorithm 6.1.1 is sound and complete with respect to the fixed point semantics: for all queries* $P_0\leftarrow c_0$ *and for all substitution* θ *satisfying* c_0

$$\theta P_0\in\|\mathit{Model}(\mathcal{P})\|\Longleftrightarrow\theta\in\|\mathit{Ans}(P_0,c_0)\|.$$

Theorem 6.1.3. *Algorithm 6.1.1 terminates for all queries on a program* $\mathcal{P}$ *whenever the fixed point semantics of* $\mathcal{P}$ *is finite.*

In §6.3, we will need the following two lemmas from [Tom97]:

Lemma 6.1.4. *Every substitution* $\theta\in\|\mathit{Ans}(P,c)\|$ *satisfies* c*, for all predicates* P *and constraints* c.

Lemma 6.1.5. *If* $c_1\Rightarrow c_2$ *then* $\|\mathit{Ans}(P,c_1)\|\subseteq\|\mathit{Ans}(P,c_2)\|$*, for all predicates* P *and constraints* c_1,c_2.

6.1.1 A Deterministic Variant

Algorithm 6.1.1 as presented in [Tom97] is nondeterministic, as the order in which the rules are applied is not specified. Moreover, the Answer Propagation step does not specify which c'' to choose. An actual implementation would first check whether any proof tree rooted in $\mathsf{root}(P_1,c'')$ such that $c'\Rightarrow c''$ already exists, and take all of its already computed as well as future answers to perform Answer Propagation. If no such proof tree exists yet, a new one is spawned with $\mathsf{root}(P_1,c'')$. The choice of c'' is free as long as $c'\Rightarrow c''$, so in particular c'' could be c', or simply true. We choose the first alternative as it is generally more efficient (the second is similar to bottom-up evaluation) and, more importantly, it maximises groundness of variables.

The following procedures written in pseudo-code make the general evaluation order as well as the Answer Propagation step from Algorithm 6.1.1 explicit. To compute the answer of a query $P\leftarrow c$, let $\mathit{Ans}(Q,d)$ and $\mathit{Wait}(Q,d)$ be initially undefined for all predicates Q and constraints d, except $\mathit{Ans}(P,c)=\emptyset$. Call RESOLVE-CLAUSE($\mathsf{root}(P,c)$). After the call, $\mathit{Ans}(P,c)$ will contain the answer of the query.

The procedure RESOLVE-CLAUSE starts a new proof tree from a given root node, creating body nodes from matching rules in the program. If the rule is a fact, PROJECT-ANSWER is

called on the corresponding body node, otherwise PROJECT-QUERY is called.

RESOLVE-CLAUSE(root$(P_0; c_0)$)
1 **foreach** $R \equiv P_0 \leftarrow \vec{P}, c \in \mathcal{P}$ such that $c_0 \wedge c$ is satisfiable **do**
2 **if** $\vec{P} = []$ **then** PROJECT-ANSWER(body$((P_0, c_0); []; c_0 \wedge c)$)
3 **else** PROJECT-QUERY(body$((P_0, c_0); \vec{P}; c_0 \wedge c)$)

The procedure PROJECT-QUERY takes a body node with a non-empty list of subgoals and projects its constraint onto the free variables of the first subgoal predicate. PROPAGATE-ANSWER is called on the resulting goal nodes.

PROJECT-QUERY(body$((P_0, c_0); [P_1, ..., P_n]; c_1)$)
1 **foreach** satisfiable $c \in \exists_{-P_1}(c_1)$ **do**
2 PROPAGATE-ANSWER(goal$((P_0, c_0); (P_1, c); [P_2, ..., P_n]; c_1)$)

PROJECT-ANSWER deals with body nodes with an empty list of subgoals. In this case, the constraint is projected onto the free variables of the original predicate, and PROCESS-ANSWER is called on the resulting answer nodes.

PROJECT-ANSWER(body$((P_0, c_0); []; c_1)$)
1 **foreach** satisfiable $c \in \exists_{-P_0}(c_1)$ **do**
2 PROCESS-ANSWER(ans$((P_0, c_0); c)$)

The PROCESS-ANSWER procedure takes an answer node and updates the *Ans* function if the answer is not already subsumed by an older answer. The *Wait* function keeps track of the goal nodes that wait for this answer. These goal nodes are combined with the answer to form a new body node that is then further processed by PROJECT-ANSWER or PROJECT-QUERY, depending on whether it contains any further subgoals.

PROCESS-ANSWER(ans$((P_0, c_0); c)$)
1 **if** c is not subsumed by a constraint in $Ans(P_0, c_0)$ **then**
2 $Ans(P_0, c_0) := Ans(P_0, c_0) \cup \{c\}$;
3 **foreach** goal$((Q_0, d_0); (P_0, d); \vec{Q}; d_1) \in Wait(P_0, c_0)$
4 such that $c \wedge d_1$ is satisfiable **do**
5 **if** $\vec{Q} = []$ **then** PROJECT-ANSWER(body$((Q_0, d_0); []; c \wedge d_1)$)
6 **else** PROJECT-QUERY(body$((Q_0, d_0); \vec{Q}; c \wedge d_1)$)

PROPAGATE-ANSWER deals with newly created goal nodes. If a proof tree that could supply the answers for the goal already exists then the goal is put onto the waiting list of that proof tree. The goal will thus receive future answers from that proof tree. Furthermore, all existing answers are propagated to the goal node to form new body nodes that are then further processed by PROJECT-ANSWER or PROJECT-QUERY, depending on whether they

contain any further subgoals.

```
PROPAGATE-ANSWER(goal((P_0,c_0);(P_1,d_0);P⃗;c_1))
01  if there exists (P_1,d_1) ∈ Dom(Ans) such that d_0 ⇒ d_1 then
02    Wait(P_1,d_1) :=
03        Wait(P_1,d_1) ∪ {goal((P_0,c_0);(P_1,d_0);P⃗;c_1)};
04    foreach a ∈ Ans(P_1,d_1) such that a ∧ c_1 is satisfiable do
05      if P⃗ = [] then PROJECT-ANSWER(body((P_0,c_0);[];a ∧ c_1))
06      else PROJECT-QUERY(body((P_0,c_0);P⃗;a ∧ c_1))
07  else
08    Ans(P_1,d_0) := ∅;
09    Wait(P_1,d_0) := {goal((P_0,c_0);(P_1,d_0);P⃗;c_1)};
10    RESOLVE-CLAUSE(root(P_1;d_0))
```

We will now discuss how this algorithm can be modified to evaluate CASSANDRA policy queries.

6.2 Evaluation Algorithm

CASSANDRA differs from Datalog^C in two aspects relevant to query evaluation: aggregation and remote predicates. Due to the restriction of aggregation-safety, the evaluation of predicates defined by aggregation rules can be performed by a separate procedure AGGREGATE that does not interfere with the evaluation of other goals.

Algorithm 6.1.1 can be modified to handle CASSANDRA's prefixed predicates. First of all, the location and issuer prefixes can be treated as additional predicate parameters. If the location of the current goal predicate is instantiated to the location of the rule, the algorithm acts as before. However, if the location is instantiated to a different entity and the goal is not subsumed by some older goal, a new proof tree for this remote goal is spawned at the remote location. This proof tree will contain the additional corresponding canReqCred goal.

Definition 6.2.1. *If c grounds e_{loc} to E, and e_{loc} is the location of a predicate, then the* location of the predicate with respect to *c is E:*

$$Loc(e_{loc}\Diamond e_{iss}.p(\vec{x}),c) = E \text{ if } c \Rightarrow e_{loc} = E, \text{ and undefined, otherwise.}$$

Procedure calls are tagged with a location to indicate where it is called. For example, $E\Diamond$RESOLVE-CLAUSE is a procedure invocation at location E. The *Ans* and *Wait* tables are now also localised; for example, $E\Diamond Ans$ is the answer function at location E.

The procedures now also have an additional parameter to keep track of who is requesting the answers. If during evaluation at E_{loc} a remote goal $(P_0;c_0)$ is encountered, say $Loc(P_0,c_0) = E_{rem}$, then $E_{rem}\Diamond$RESOLVE-CLAUSE$(E_{loc}, \mathsf{root}(P_0;c_0))$ is called. The answers produced at E_{rem} will be sent back to, stored and further processed at E_{loc}.

To compute the answer of a query $P \leftarrow c$ where $P \equiv E_{loc}\Diamond E_{iss}.p_0(\vec{e}_0)$, let $E\Diamond Ans(Q,d)$ and $E\Diamond Wait(Q,d)$ be initially undefined for all entities E, predicates Q and constraints d except $E_{loc}\Diamond Ans(P,c) = \emptyset$. Call $E_{loc}\Diamond$RESOLVE-CLAUSE$(E_{loc}, \mathsf{root}(P,c))$. After the call returns, $E_{loc}\Diamond Ans(P,c)$ will contain the answer of the query.

The modified RESOLVE-CLAUSE procedure calls AGGREGATE to deal with aggregation rules.

For non-aggregate rules there are two cases. If the requester is local, the resulting body node is processed as before. Otherwise, i.e. if the requester is remote, the subgoals of the body node are extended with a matching canReqCred goal that checks whether (and under which restrictions) the policy allows the requester to perform this query.

$E_{loc}\Diamond$RESOLVE-CLAUSE$(E_{req}, \mathsf{root}(P_0; c_0))$
1 **foreach** $R \equiv P_0 \leftarrow \vec{P}, c \in \mathcal{P}$ such that $c_0 \wedge c$ is satisfiable **do**
2 **if** R is an aggregation rule **then**
3 $E_{loc}\Diamond$AGGREGATE$(E_{req}, (P_0, c_0), R)$
4 **else if** $E_{req} = E_{loc}$ **then**
5 $E_{loc}\Diamond$PROJECT$(E_{req}, \mathsf{body}((P_0, c_0); \vec{P}; c_0 \wedge c))$
6 **else**
7 $E_{loc}\Diamond$PROJECT$(E_{req}, \mathsf{body}((P_0, c_0); [\mathsf{canReqCred}(E_{req}, P_0), \vec{P}]; c_0 \wedge c))$

The AGGREGATE procedure assumes that the aggregation control parameters $\vec{y}$ are grounded by $c \wedge c_0$. Then it computes the set X_R of all different values that x can assume, applies the aggregation operator (i.e. in the case of count, it computes the size of the set) and calls PROCESS-ANSWER at the requester's location on the resulting answer nodes.

$E_{loc}\Diamond$AGGREGATE$(E_{req}, (P_0, c_0), R)$
1 **match** R **with** $E_{loc}\Diamond E_{loc}.p_0(\mathsf{aggOp}\langle x\rangle, \vec{y}) \leftarrow E_{loc}\Diamond e'.p_1(\vec{z}), c$ **in**
2 **match** P_0 **with** $E_{loc}\Diamond E_{loc}.p_0(w, \vec{y})$ **in**
3 **foreach** $c_0' \in \exists_{-P_0}(c_0)$ **do**
4 **let** $X_R = \{\ K_x \mid E_{loc}\Diamond E'.p_1(\vec{z}) \leftarrow \vec{z} = \vec{K} \in \mathcal{P},$
5 $\vec{z} = \vec{K} \Rightarrow x = K_x,$
6 $c_0' \wedge c \wedge \vec{z} = \vec{K} \wedge e' = E'$ is satisfiable $\}$ **in**
7 $E_{req}\Diamond$PROCESS-ANSWER$(\mathsf{ans}(P_0, c_0); c_0' \wedge w = \mathsf{aggOp}(X_R))$

PROJECT works in the same way as PROJECT-ANSWER and PROJECT-QUERY from §6.1.1.

$E_{loc}\Diamond$PROJECT$(E_{req}, \mathsf{body}((P_0, c_0); \vec{P}, c_1))$
1 **if** $\vec{P} = []$ **then**
2 **foreach** satisfiable $c \in \exists_{-P_0}(c_1)$ **do**
3 $E_{req}\Diamond$PROCESS-ANSWER$(\mathsf{ans}((P_0, c_0); c))$
4 **else**
5 **foreach** satisfiable $c \in \exists_{-P_1}(c_1)$ **do**
6 $E_{loc}\Diamond$PROPAGATE-ANSWER$(E_{req}, \mathsf{goal}((P_0, c_0); (P_1, c); \vec{P}; c_1))$

The PROCESS-ANSWER procedure is also unchanged apart from the additional location tags.

$E_{loc}\Diamond$PROCESS-ANSWER$(\mathsf{ans}(P_0, c_0), c)$
1 **if** c is not subsumed by a constraint in $E_{loc}\Diamond Ans(P_0, c_0)$ **then**
2 $E_{loc}\Diamond Ans(P_0, c_0) := E_{loc}\Diamond Ans(P_0, c_0) \cup \{c\};$
3 **foreach** $(E_{req}, \mathsf{goal}((Q_0, d_0); (P_0, d); \vec{Q}; d_1)) \in E_{loc}\Diamond Wait(P_0, c_0)$
4 such that $c \wedge d_1$ is satisfiable **do**
5 $E_{loc}\Diamond$PROJECT$(E_{req}, \mathsf{body}((Q_0, d_0); \vec{Q}; c \wedge d_1)$

The values of the *Wait* function, updated by the PROPAGATE-ANSWER procedure, now also contain the requesting entity. To process the goal node, a new proof tree may have to

be spawned, as before, but now it may be spawned at a remote location. The location is determined by $Loc(P_1, d_0)$ in Line 9. Of course, this only works if the location is defined at that point. We will later use static groundness analysis to enforce this condition. The requester of the new proof tree is set to E_{loc}.

$E_{loc}\Diamond$PROPAGATE-ANSWER$(E_{req}, \mathsf{goal}((P_0, c_0); (P_1, d_0); \vec{P}; c_1))$
1 **if** there exists $(P_1, d_1) \in Dom(E_{loc}\Diamond Ans)$ such that $d_0 \Rightarrow d_1$ **then**
2 $\quad E_{loc}\Diamond Wait(P_1, d_1) :=$
3 $\quad\quad E_{loc}\Diamond Wait(P_1, d_1) \cup (E_{req}, \mathsf{goal}((P_0, c_0); (P_1, d_0); \vec{P}; c_1))$;
4 $\quad$ **foreach** $a \in Ans(P_1, d_1)$ such that $a \wedge c_1$ is satisfiable **do**
5 $\quad\quad E_{loc}\Diamond$PROJECT$(E_{req}, \mathsf{body}((P_0, c_0); \vec{P}; a \wedge c_1))$
6 **else**
7 $\quad E_{loc}\Diamond Ans(P_1, d_0) := \emptyset$;
8 $\quad E_{loc}\Diamond Wait(P_1, d_0) := \{(E_{req}, \mathsf{goal}((P_0, c_0); (P_1, d_0); \vec{P}; c_1))\}$;
9 $\quad Loc(P_1, d_0)\Diamond$RESOLVE-CLAUSE$(E_{loc}, \mathsf{root}(P_1; d_0))$

For both aggregation and remote predicates, static groundness analysis is employed to simplify evaluation. Groundness analysis can also be used in CASSANDRA to enforce other useful conditions. This is discussed in the following section.

6.3 Groundness Analysis

Groundness analysis is a very useful static program analysis that can be used on CASSANDRA policies for three different purposes:

- If, in the process of evaluating a query, a goal predicate has an unconstrained location, the evaluation engine cannot figure out where to deduce it from. Therefore, the location of a body predicate is required to be ground by the time all body predicates preceding it in the rule have been solved.
- There are useful constraint domains such as $\mathcal{C}_0$ that are very difficult to prove to be constraint compact or are not constraint compact at all. By requiring that certain constraint constructs are ground during answer projection (which is the only point where exact existential quantifier elimination is needed), we can restrict the constraint domain to a constraint compact fragment during run-time, thus preserving the termination properties. It also simplifies the implementation of such constraint domains if it is known in advance that certain constructs will always be ground. In the case of $\mathcal{C}_0$, we require that built-in-function arguments and set constraints are ground during answer projection, i.e. after all body predicates of the respective rule have been solved. This reduces the constraints to the simpler constraint domain $\mathcal{C}_{\neq,<}$ for which we have proven constraint compactness.
- The requirement of aggregation-safety on aggregation rules is rather strict (cf. Definition 4.4.2). It is often useful to have control parameters in the head of an aggregation rule that do not occur in the parameters of the body predicate. Chapters 8 and 9 provide examples of such rules. Such rules reduce to aggregation-safe rules if these "unsafe" control parameters are always ground during run-time. In fact, the algorithm given in §6.2 requires that all control parameters of aggregation rules are ground during run-time, i.e. whenever an aggregate body predicate is to be solved. This requirement greatly simplifies implementation and enhances efficiency.

In order to verify such requirements, the analysis can exploit the fact that in CASSANDRA, queries are only ever issued by the access control engine, and only with particular groundness patterns. For example, a query involving permits, canActivate, canDeactivate or isDeactivated will always be fully instantiated. However, in queries involving canReqCred$(E_r, E_{iss}.p(\vec{x}))$, the arguments $\vec{x}$ may be uninstantiated, so the access control engine may also issue queries involving $E_{iss}.p(\vec{x})$ with uninstantiated $\vec{x}$, for all predicate names p occurring in a head of some canReqCred rule (see Chapter 7).

For example, suppose `TheZoo`'s policy had the following rule:

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{Monkey}(age)) \leftarrow \\ &\quad x \Diamond \texttt{TheZoo}.\mathsf{monkeyCert}(age)\end{aligned}$$

Here, the variable x is required to be ground before the first body predicate is processed, since it is used as the location of a predicate. In other words, we have to check that whenever a query is issued by the access control engine, all answer nodes to the isGround001 predicate acting as "groundness checkpoint" in the rule below will be ground. Such groundness checkpoints can be inserted whenever the groundness of a variable needs to checked.

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{Monkey}(age)) \leftarrow \\ &\quad \mathsf{isGround001}(\mathsf{x}), \\ &\quad x \Diamond \texttt{TheZoo}.\mathsf{monkeyCert}(age) \\ &\mathsf{isGround001}(\mathsf{x}) \leftarrow \mathsf{true}\end{aligned} \tag{6.1}$$

Given that there are no other rules, the groundness of x can be deduced by groundness analysis under the assumption that the arguments to the canActivate head of the rule will be fully instantiated. However, if we added the following rule, the check would fail, since Rule 6.1 would be called with an uninstantiated x whenever someone tried to activate the role `SomeRole`:

$$\begin{aligned}&\mathsf{canActivate}(y, \texttt{SomeRole}()) \leftarrow \\ &\quad \mathsf{canActivate}(x, \texttt{Monkey}(age))\end{aligned}$$

Similarly, if there were a rule

$$\mathsf{canReqCred}(y, \texttt{TheZoo}.\mathsf{canActivate}(x, \texttt{Monkey}(age))) \leftarrow age \geq 18$$

then x may also be uninstantiated at the groundness checkpoint in Rule 6.1, because the access control engine may issue a canReqCred query with uninstantiated arguments.

The following section describes an algorithm that can statically check the groundness of nodes in an SLG^C forest for queries with a given groundness pattern.

6.3.1 Positive Boolean Functions for Groundness Analysis

Pos, the set of positive boolean functions, has previously been proposed as an abstract interpretation domain [CC77, Cou96] for groundness analysis on logic programs and constraint logic programs [MS93, BS93]. A boolean function is *positive* if it is satisfied by the assignment of true to all its variables. The positive boolean functions are closed under conjunction, disjunction and existential quantifier elimination. Subsumption of boolean functions is equivalent to logical implication. We will present a proof that the SLG^C-resolution algorithm can be used to perform *Pos*-based groundness analysis on Datalog^C programs.

This can simply be done by mapping every constraint c in the program into an "abstract constraint" $c^\sharp$ in the *Pos* domain that contains groundness information about c. Then, any query $P_0 \leftarrow c_0$ can be mapped into an "abstract query" $P_0 \leftarrow c_0^\sharp$. The SLG^C proof forest for the abstract query will then contain all information about run-time groundness of the original query. The mapping we will be using is a fragment of the one mentioned in [BS93]:

Definition 6.3.1. *A C_0-constraint c is mapped to a Pos-constraint $c^\sharp$ as follows:*

- $(x = k)^\sharp = x$ *if k is a ground expression*
- $(x = y)^\sharp = x \longleftrightarrow y$
- $(c \wedge c')^\sharp = c^\sharp \wedge c'^\sharp$
- $(c \vee c')^\sharp = c^\sharp \vee c'^\sharp$
- $c^\sharp = \mathsf{true}^\sharp$ *if c does not match any of the above*

The mapping is supposed to convey groundness information of the original constraint. For example, the intuitive reading of $x \wedge y$ is "x and y are both ground", whereas $x \longleftrightarrow y$ means "x is ground iff y is ground". The mapping is only approximate, though: for example, $1 < x < 3$ grounds x as it is equivalent to $x = 2$ but $(1 < x < 3)^\sharp = \mathsf{true}^\sharp$ which conveys no groundness information about x whatsoever. The mapping can be made more precise, if required, as long as it is sound in the sense that if fulfils the following requirement: $c^\sharp \Rightarrow x$ implies that c grounds x.

The main result, Theorem 6.3.9, states that the answer nodes in the SLG^C forest of an abstract query contain correct groundness information about the SLG^C forest of the original query. The correctness of the analysis depends on a condition on constraint domains which we call $\exists/\sharp$-commutativity[2]:

Definition 6.3.2. **($\exists/\sharp$-commutative)** *A constraint domain C is* $\exists/\sharp$-commutative *if for every C-constraint c*

$$(\exists x(c))^\sharp \Rightarrow \exists x(c^\sharp).$$

We will later prove that C_0 is an example of a $\exists/\sharp$-commutative constraint domain. First we need to prove some auxiliary lemmas that lead to Theorem 6.3.9. Lemmas 6.3.3 and 6.3.4 prove general properties about $\exists$ and $\sharp$ that hold for all constraint domains. Lemma 6.3.5 states a general property about SLG^C resolution.

Lemma 6.3.3. *If $c \Rightarrow c'$ then $\exists x(c) \Rightarrow \exists x(c')$.*

Proof. Suppose $c \Rightarrow c'$, and $\theta \models c_x \in \exists x(c)$. Then there exists a constant k such that $\theta \models c[k/x]$, and thus also $\theta \models c'[k/x]$. Then there is also a constraint $c'_x \in \exists x(c')$ such that $\theta \models c'_x$. Hence $c_x \Rightarrow c'_x$. □

Lemma 6.3.4. *Let c be a constraint from a $\exists/\sharp$-commutative constraint domain. If $c^\sharp \Rightarrow \gamma$ then $(\exists x(c))^\sharp \Rightarrow \exists x(\gamma)$.*

Proof. Suppose $c^\sharp \Rightarrow \gamma$. By $\exists/\sharp$-commutativity, $(\exists x(c))^\sharp \Rightarrow \exists x(c^\sharp)$. By Lemma 6.3.3, $\exists x(c^\sharp) \Rightarrow \exists x(\gamma)$. □

[2] If C_1 and C_2 are sets of constraints, $C_1 \Rightarrow C_2$ is shorthand for "for all $c_1 \in C_1$ there exists $c_2 \in C_2$ such that $c_1 \Rightarrow c_2$. If C is a set of constraints, then $C^\sharp$ is shorthand for $\{c^\sharp \mid c \in C\}$.

Lemma 6.3.5. *Let c, c' be constraints and G a predicate with $Fv(c) \cup Fv(c') \subseteq Fv(G)$. If $c \Rightarrow c'$ then*[3] $\|Ans(G,c)\| = \|c \wedge Ans(G,c')\|$.

Proof. First suppose $\theta \in \|Ans(G,c)\|$. Since $c \Rightarrow c'$, $\|Ans(G,c)\| \subseteq \|Ans(G,c')\|$ by Lemma 6.1.5. Furthermore, by Lemma 6.1.4, $\theta \models c$, hence $\theta \in \|c \wedge Ans(G,c')\|$.

For the other direction, suppose $\theta \in \|c \wedge Ans(G,c')\|$. By Theorem 6.1.2, $\theta G \in ||Model(\mathcal{P})||$. Furthermore, $\theta \models c$, so again by Theorem 6.1.2, $\theta \in \|Ans(G,c)\|$. □

Next, we define a subsumption relation between $\mathcal{C}$-proof trees T and *Pos*-proof trees Υ. Intuitively, the former T is subsumed by Υ if the groundness information in the latter is at most as "definite" as the groundness information in T, if all constraints in T are mapped to *Pos*-constraints. This relation is central in the proof of Theorem 6.3.9.

Definition 6.3.6. *A proof tree T is* subsumed *by a Pos-proof tree Υ if for all Pos-substitutions θ and for all nodes*

- $\mathsf{root}(P;c) \in T$*: if* $\theta \models c^\sharp$ *then there exists some* $\mathsf{root}(P;\gamma) \in \Upsilon$ *such that* $\theta \models \gamma$.
- $\mathsf{body}((P,c_0);\vec{Q};c) \in T$*: if* $\theta \models c^\sharp$ *then there exists some* $\mathsf{body}((P,\gamma_0);\vec{Q};\gamma) \in \Upsilon$ *such that* $\theta \models \gamma$.
- $\mathsf{goal}((P,c_0);(Q,c);\vec{R};c') \in T$*: if* $\theta \models c^\sharp$ *then there exists some* $\mathsf{goal}((P,\gamma_0);(Q,\gamma);\vec{R};\gamma') \in \Upsilon$ *such that* $\theta \models \gamma$*, and if* $\theta \models c'^\sharp$ *then there exists some* $\mathsf{goal}((P,\gamma_0);(Q,\gamma);\vec{R};\gamma') \in \Upsilon$ *such that* $\theta \models \gamma'$.
- $\mathsf{ans}((P,c_0);c) \in T$*: if* $\theta \models c^\sharp$ *then there exists some* $\mathsf{ans}((P,\gamma_0);\gamma) \in \Upsilon$ *such that* $\theta \models \gamma$.

Now we can prove the two main Lemmas 6.3.7 and 6.3.8 about SLG$^{\mathcal{C}}$ resolution on *Pos*-translated programs. In essence, they state that the abstract proof forest is at most as "definite" about groundness as its original counterpart. This fact is used in the proof of Theorem 6.3.9: if the abstract proof forest implies that a variable x is ground at some point (that is a very "definite" statement), it must actually be ground in the original proof forest.

Lemma 6.3.7. *Let the constraint domain be $\exists/\sharp$-commutative. Let F be a proof forest containing a proof tree T with root node* $\mathsf{root}(G;\hat{c})$*. If $\hat{\gamma}$ is a Pos-constraint such that $\hat{c}^\sharp \Rightarrow \hat{\gamma}$ then T is subsumed by the root tree Υ of the proof forest Φ of the query $(G \leftarrow \hat{\gamma})$.*

Proof. By induction on the stages of the SLG$^{\mathcal{C}}$ algorithm running on the query $(G \leftarrow \hat{c})$. The statement is vacuously true at step 0. Consider the node N of a tree T in F created at step $n+1$. If N is a root node, the statement is trivially true. We need to consider the three remaining cases.

(1) Suppose N is of the form $\mathsf{body}((G,\hat{c});\vec{B};c)$. There are two subcases. Either N's parent is $\mathsf{root}(G;\hat{c})$ where $c = \hat{c} \wedge d$. Let $\theta \models c^\sharp$. Then we also have $\theta \models \hat{c}^\sharp$ and $\theta \models d^\sharp$. Let $\hat{\gamma}$ be any constraint such that $\hat{c}^\sharp \Rightarrow \hat{\gamma}$. Then by the definition of RESOLVE-CLAUSE, the root of Υ, $\mathsf{root}(G;\hat{\gamma})$, has a child node $\mathsf{body}((G,\hat{\gamma});\vec{B};\hat{\gamma} \wedge d^\sharp)$, and $\theta \models \hat{\gamma} \wedge d^\sharp$.

In the second case, N's parent is $\mathsf{goal}((G,\hat{c});(B;c');\vec{B};d)$. By PROPAGATE-ANSWER, N is waiting for some tree T' with $\mathsf{root}(B,c'')$ such that $c' \Rightarrow c''$. Let $\theta \models c$. But θ also satisfies $d^\sharp$ as $c = d \wedge a$, for some $a \in Ans(B,c'')$. So by the induction hypothesis, Υ has a node $\nu = \mathsf{goal}((G,\hat{\gamma});(B;\gamma');\vec{B};\delta)$ such that $\theta \models \delta$. By the answer propagation rule, ν waits for some tree Υ' with $\mathsf{root}(B,\gamma'')$ where $\gamma' \Rightarrow \gamma''$.

[3] If C is a set of constraints, $c \wedge C$ is shorthand for $\{c \wedge c' \mid c' \in C\}$.

Also by the induction hypothesis, all nodes of T' with age $\leq n+1$ are subsumed by nodes of the root tree of the proof forest for the query $(B \leftarrow \mathsf{true}^\sharp)$. So, in particular, that root tree has a node $\mathsf{ans}((B,\mathsf{true}^\sharp),\alpha)$ with $\theta \models \alpha$, since $\theta \models a^\sharp$. By the answer propagation rule, ν's child nodes are all of the form $\mathsf{body}((G,\hat{\gamma});\vec{B};\delta \wedge \overline{\alpha})$ where $\overline{\alpha} \in \mathit{Ans}(B,\gamma'')$. But since $\delta \Rightarrow \gamma''$, Lemma 6.3.5 implies that $\|\delta \wedge \mathit{Ans}(B,\gamma'')\| = \|\delta \wedge \mathit{Ans}(B,\mathsf{true}^\sharp)\|$. Therefore, and since $\theta \in \|\delta \wedge \mathit{Ans}(B,\mathsf{true}^\sharp)\|$, we also have $\theta \in \|\delta \wedge \mathit{Ans}(B,\gamma'')\|$. Hence, by the answer propagation rule, ν has a child node $\mathsf{body}((G,\hat{\gamma});\vec{B};\delta \wedge \overline{\alpha})$ such that $\theta \models \delta \wedge \overline{\alpha}$.

(2) Suppose N is of the form $\mathsf{goal}((G,\hat{c});(B;c');\vec{B};c)$. By the query projection rule, it has a parent node of the form $\mathsf{body}((G,\hat{c});B :: \vec{B};c)$, and $c' \in \exists_B(c)$. By the induction hypothesis, the root tree Υ of the proof forest of the query $(G \leftarrow \hat{\gamma})$ contains, for all $\theta \models c$, a node $\mathsf{body}((G,\hat{\gamma});B :: \vec{B};\gamma_\theta)$ such that $\theta \models \gamma_\theta$, hence

$$c^\sharp \Rightarrow \bigvee_{\theta \models c^\sharp} \gamma_\theta.$$

From this and the query projection rule, it firstly follows that for all $\theta \models c^\sharp$, Υ also has a node of the form $\mathsf{goal}((G,\hat{\gamma});(B;\gamma');\vec{B};\gamma_\theta)$ such that $\theta \models \gamma_\theta$. Secondly, by Lemma 6.3.4, if $\theta' \models c'^\sharp$ then $\theta' \models \gamma'$ for some

$$\gamma' \in \exists_B(\bigvee_{\theta \models c^\sharp} \gamma_\theta) = \bigcup_{\theta \models c^\sharp} \exists_B(\gamma_\theta).$$

So by the query projection rule, Υ also has a node $\mathsf{goal}((G,\hat{\gamma});(B;\gamma');\vec{B};\gamma)$ such that $\theta' \models \gamma'$, for all $\theta' \models c'^\sharp$. Hence all nodes of T with age $\leq n+1$ are subsumed by Υ.

(3) Suppose N is of the form $\mathsf{ans}((G,\hat{c});c)$. Let $\theta \models c^\sharp$. By the answer projection rule, it has a parent node of the form $\mathsf{body}((G,\hat{c});;c')$, and $c \in \exists_G(c')$. By the induction hypothesis, Υ contains, for all $\theta' \models c'^\sharp$, a node $\mathsf{body}((G,\hat{\gamma});;\gamma'_{\theta'})$ such that $\theta' \models \gamma'_{\theta'}$, hence

$$c'^\sharp \Rightarrow \bigvee_{\theta' \models c'^\sharp} \gamma'_{\theta'}.$$

By Lemma 6.3.4 and from $\theta \models c^\sharp$ it follows that $\theta \models \gamma$ for some

$$\gamma \in \exists_G(\bigvee_{\theta' \models c'^\sharp} \gamma'_{\theta'}) = \bigcup_{\theta' \models c'^\sharp} \exists_G(\gamma'_{\theta'}).$$

So by the answer projection rule, Υ contains a node $\mathsf{body}((G,\hat{\gamma});;\gamma')$ with a child node $\mathsf{ans}((G,\hat{\gamma});\gamma)$ such that $\theta \models \gamma$, as required. □

Lemma 6.3.8. *Let the constraint domain be $\exists/\sharp$-commutative. Let F and Φ be proof forests for the queries $(\hat{G} \leftarrow \hat{c})$ and $(\hat{G} \leftarrow \hat{\gamma})$, respectively, with $\hat{c}^\sharp \Rightarrow \hat{\gamma}$. Then every tree T in F is subsumed by a tree Υ in Φ.*

Proof. By induction on the rewriting steps of the algorithm running on the query $(\hat{G} \leftarrow \hat{c})$. The statement is vacuously true at step 0. Consider the node N of a tree T in F created at step $n+1$. We need to consider four cases.

(1) Suppose N is of the form $\mathsf{root}(G;c)$, and $\theta \models c^\sharp$. Then T must have been spawned by some older $\mathsf{goal}((G',c');(G;c);\vec{B};c'')$ in F. By the induction hypothesis, there exists a $\mathsf{goal}((G',\gamma');(G;\gamma);\vec{B};\gamma'')$ in Φ such that $\theta \models \gamma$. By the answer propagation rule, this node

waits for a tree Υ with $\mathsf{root}(G;\delta)$ where $\gamma \Rightarrow \delta$. So $\theta \models \delta$, and since at step $n+1$, T contains only N, all nodes of T with age $\leq n+1$ are subsumed by Υ.

(2) Suppose N is of the form $\mathsf{body}((G,c_0);\vec{B};c)$, and $\theta \models c^\sharp$. There are two subcases. Either N's parent is $\mathsf{root}(G;c_0)$ and $c = c_0 \wedge d$, for some d. Note that θ satisfies $c_0^\sharp$. Therefore, by the induction hypothesis, there is a tree Υ in Φ with $\mathsf{root}(G;\gamma_0)$ such that $\theta \models \gamma_0$. Since θ also satisfies $d^\sharp$, it satisfies $\gamma_0 \wedge d^\sharp$. Therefore, by the clause resolution rule, Υ has a node $\mathsf{body}((G,\gamma_0);\vec{B};\gamma_0 \wedge d^\sharp)$.

In the second case, N's parent is $\mathsf{goal}((G,c_0);(B;c');\vec{B};d)$. By the answer propagation rule, N is waiting for some tree T' with $\mathsf{root}(B,c'')$ such that $c' \Rightarrow c''$. Since $c = d \wedge a$ for some $a \in \mathit{Ans}(B,c'')$, θ also satisfies $d^\sharp$. So by the induction hypothesis, there is a tree Υ in Φ with a node $\nu = \mathsf{goal}((G,\gamma_0);(B;\gamma');\vec{B};\delta)$ such that $\theta \models \delta$. By the answer propagation rule, ν waits for some tree Υ' with $\mathsf{root}(B,\gamma'')$ where $\gamma' \Rightarrow \gamma''$. Therefore, ν's children are of the form $\mathsf{body}((G,\gamma_0);\vec{B};\delta \wedge \overline{\alpha})$ where $\overline{\alpha} \in \mathit{Ans}(B,\gamma'')$. It suffices to show $\theta \models \delta \wedge \overline{\alpha}$ for one of these child nodes.

By Lemma 6.3.7, T' is subsumed by the root tree Υ'' of a proof forest for the query $(B \leftarrow \mathsf{true}^\sharp)$. So since θ satisfies a and $\mathsf{ans}((B,\mathsf{true}^\sharp),a) \in T'$, $\theta \in \|\mathit{Ans}(B,\mathsf{true}^\sharp)\|$. As θ also satisfies δ, $\theta \in \|\delta \wedge \mathit{Ans}(B,\mathsf{true}^\sharp))\|$. But since $\gamma'' \Rightarrow \mathsf{true}^\sharp$ and by Lemma 6.3.5, $\|\mathit{Ans}(B,\gamma'')\| = \|\gamma'' \wedge \mathit{Ans}(B,\mathsf{true}^\sharp)\|$, and furthermore since $\delta \Rightarrow \gamma' \Rightarrow \gamma''$, $\|\delta \wedge \mathit{Ans}(B,\gamma'')\| = \|\delta \wedge \mathit{Ans}(B,\mathsf{true}^\sharp)\|$. So $\theta \in \|\delta \wedge \mathit{Ans}(B,\gamma'')\|$. Therefore ν has a child node $\mathsf{body}((G,\gamma_0);\vec{B};\delta \wedge \overline{\alpha})$ such that $\theta \models \delta \wedge \overline{\alpha}$.

(3) Suppose N is of the form $\mathsf{goal}((G,c_0);(B;c');\vec{B};c)$. By the query projection rule, it has a parent node of the form $\mathsf{body}((G,c_0);B :: \vec{B};c)$, and $c' \in \exists_B(c)$. By the induction hypothesis, there exists a tree Υ in Φ such that for all $\theta \models c^\sharp$, Υ has a node $\mathsf{body}((G,\gamma_0);B :: \vec{B};\gamma_\theta)$ such that $\theta \models \gamma_\theta$, hence

$$c^\sharp \Rightarrow \bigvee_{\theta \models c^\sharp} \gamma_\theta.$$

From this and the query projection rule, it firstly follows that for all $\theta \models c^\sharp$, Υ also has a node of the form $\mathsf{goal}((G,\gamma_0);(B;\gamma');\vec{B};\gamma_\theta)$ such that $\theta \models \gamma_\theta$. Secondly, by Lemma 6.3.4, if $\theta' \models c'^\sharp$ then $\theta' \models \gamma'$ for some

$$\gamma' \in \exists_B(\bigvee_{\theta \models c^\sharp} \gamma_\theta) = \bigcup_{\theta \models c^\sharp} \exists_B(\gamma_\theta).$$

So by the query projection rule, for all $\theta' \models c'$, Υ contains a node $\mathsf{body}((G,\gamma_0);B :: \vec{B};\gamma)$ with a child node $\mathsf{goal}((G,\gamma_0);(B;\gamma');\vec{B};\gamma)$ such that $\theta' \models \gamma'$. Hence all nodes of T with age $\leq n+1$ are subsumed by Υ.

(4) Suppose N is of the form $\mathsf{ans}((G,c_0);c)$. Let $\theta \models c^\sharp$. By the answer projection rule, it has a parent node of the form $\mathsf{body}((G,c_0);;c')$, and $c \in \exists_G(c')$. By the induction hypothesis, there exists a tree Υ in Φ such that for all $\theta' \models c'^\sharp$ it contains a node $\mathsf{body}((G,\gamma_0);;\gamma'_{\theta'})$ such that $\theta' \models \gamma'_{\theta'}$, hence

$$c'^\sharp \Rightarrow \bigvee_{\theta' \models c'^\sharp} \gamma'_{\theta'}.$$

By Lemma 6.3.4 and from $\theta \models c^\sharp$ it follows that $\theta \models \gamma$ for some

$$\gamma \in \exists_G(\bigvee_{\theta' \models c'^\sharp} \gamma'_{\theta'}) = \bigcup_{\theta' \models c'^\sharp} \exists_G(\gamma'_{\theta'}).$$

So by the answer projection rule, Υ contains a node $\mathsf{body}((G,\gamma_0);;\gamma')$ with a child node $\mathsf{ans}((G,\gamma_0);\gamma)$ such that $\theta \models \gamma$, as required. □

Theorem 6.3.9. *Let the constraint domain be $\exists/\sharp$-commutative. Let F be the proof forest of the query $(G \leftarrow c)$, Φ be the proof forest of the query $(G \leftarrow c^\sharp)$, and P be some predicate. If for all α_0, all answer nodes $\mathsf{ans}((P,\alpha_0);\alpha)$ of Φ are such that $\alpha \Rightarrow x$ then a grounds x for all answer nodes $\mathsf{ans}((P,a_0);a)$ of F, for all a_0.*

Proof. Let $\mathsf{ans}((P,a_0);a)$ be a node in F. Suppose $\theta \models a^\sharp$. Then by Lemma 6.3.8, there exists a node $\mathsf{ans}((P,\alpha_0);\alpha)$ in Φ such that $\theta \models \alpha$. Since $\alpha \Rightarrow x$, θ must also satisfy x. Therefore $a^\sharp \Rightarrow x$, or in other words, a grounds x. □

Now we have proved that we can use the SLG$^\mathcal{C}$ algorithm to perform groundness analysis. The correctness of the analysis is dependent on the constraint domain being $\exists/\sharp$-commutative. We next prove that $\mathcal{C}_0$ enjoys this property.

Lemma 6.3.10. *Let c be a $\mathcal{C}_0$ constraint. Then $c[k/x]^\sharp = c^\sharp[\mathsf{true}^\sharp/x]$*

Proof. By induction on the structure of c. It is easy to show that the statement holds for atomic constraints. For instance, if c is an atomic constraint of the form $x = y$, then $c[k/x]^\sharp = y$, and also $c^\sharp[\mathsf{true}^\sharp/x] = (x \longleftrightarrow y)[\mathsf{true}^\sharp/x] = y$.

If c is of the form $c_1 \wedge c_2$ then $c[k/x]^\sharp = c_1[k/x]^\sharp \wedge c_2[k/x]^\sharp$. By the induction hypothesis, this is equal to $c_1^\sharp[\mathsf{true}^\sharp/x] \wedge c_2^\sharp[\mathsf{true}^\sharp/x] = c^\sharp[\mathsf{true}^\sharp/x]$. Similarly, the statement holds if c is of the form $c_1 \vee c_2$. □

Theorem 6.3.11. *$\mathcal{C}_0$ is $\exists/\sharp$-commutative.*

Proof. By structural induction on c. If x is not free in c, the statement trivially holds, so suppose x is free in c. If c is atomic, $\exists x(c^\sharp) = \mathsf{true}^\sharp$, so the implication always holds. Suppose c is of the form $c_1 \vee c_2$. Then $(\exists x(c_1 \vee c_2))^\sharp = (\exists x(c_1))^\sharp \cup (\exists x(c_2))^\sharp$, and by the induction hypothesis, this set is subsumed by $\exists x(c_1^\sharp) \cup \exists x(c_2^\sharp)$ which is equal to $\exists x(c_1 \vee c_2)^\sharp$.

Now suppose c is of the form $c_1 \wedge c_2$. We can assume w.l.o.g. that c is in DNF, hence c_1 and c_2 are disjunction free. Therefore, and by associativity of conjunction, we can assume w.l.o.g. that c_2 is atomic. If x is not free in c_2, $(\exists x(c_1 \wedge c_2))^\sharp = (\exists x(c_1))^\sharp \wedge c_2^\sharp$, and the statement holds by the induction hypothesis.

Suppose c_2 is of the form $x = k$ where k is a ground expression. By distributivity of $\sharp$ over $\wedge$ and $\vee$, $(\exists x(c_1 \wedge x = k))^\sharp = \mathit{DNF}(c_1[k/x]^\sharp)$. By Lemma 6.3.10, this set is equal to $\mathit{DNF}(c_1^\sharp[\mathsf{true}^\sharp/x]) = \exists x(c_1^\sharp \wedge x) = \exists x(c_1 \wedge x = k)^\sharp$.

Suppose c_2 is of the form $x = y$. By definition of $\exists$ and by distributivity of $\sharp$ over $\wedge$ and $\vee$, $(\exists x(c_1 \wedge x = y))^\sharp = \mathit{DNF}(c_1[y/x]^\sharp)$. This is equal to $\mathit{DNF}(c_1^\sharp[y/x])$, which is subsumed by $\exists x(c_1^\sharp \wedge x \longleftrightarrow y)$.

Finally, if c is a conjunction of atomic constraints all of which contain free occurrences of x and do not match any of the above cases, $c^\sharp = \mathsf{true}^\sharp$, which trivially subsumes $(\exists x(c))^\sharp$. □

To summarise, we have shown that the SLG$^\mathcal{C}$ algorithm can be used to perform static groundness analysis on programs by mapping all constraints into the domain of positive

boolean functions. The analysis is correct if the abstract mapping and existential quantifier elimination commute. We have shown that $\mathcal{C}_0$ satisfies this property, hence $\mathcal{C}_0$ policies are amenable to groundness analysis. The main uses of groundness analysis in our language are, firstly, guaranteeing groundness of predicate locations; secondly, restricting constraint domains to run-time constraint compact fragments; and thirdly, to remove some of the syntactic restrictions on aggregation rules.

Groundness analysis is not yet implemented for the current prototype of CASSANDRA (see Chapter 10), so we could not yet machine-check the large policy given in the EHR case study (Chapter 9). The policy was, however, constructed with the groundness requirements in mind, and was also manually checked afterwards. Besides, there are only relatively few rules where the restrictions would apply. We are therefore reasonably confident that the policy would pass the groundness check.

7
Policy Enforcement

CASSANDRA acts as a protective layer around a system's resources. Entities wishing to access the resources have to submit requests (such as activating or deativating a role) to the access control engine through the interface. The specification of the interface and the state-changing behaviour of the access control engine described in this chapter completes the formal model of the entire system.

Most existing research on trust management has focussed on policy specification and query evaluation and hardly on the problem of enforcing policy. Some trust management systems such as PolicyMaker [BFL96, BFK99c], KeyNote [BFK99a, BFK99b], QCM [GJ00b] or SD3 [Jim01] are explicitly designed to merely act as query evaluation engines. They leave it to the applications to invoke the trust management system, to feed it with a policy and credentials, and to formulate an appropriate query. It is also up to the application to interpret and enforce the answer for the policy request.

Some policy languages have an intended access control meaning but do not specify it completely and formally. For example, policy rules are statements governing role activation and deactivation. A formal model is described in [YMB02, BMY02], but many important details are only described informally, e.g. who can grant and revoke appointment certificates and how.

A formal specification of the access control engine is useful as it involves subtle design decisions that should be made precise to avoid ambiguities and confusion over the terms used. But most importantly, the definition of the access control engine also specifies the way in which the state is changed upon a request. Modelling the state is essential because many policies are implicitly history-dependent, as in the following examples.

- A cheque must not be authorised by the person who *initialised* it.
- A clinician may access a patient's data if consent *has been given*.
- A user may act in the manager role if they *have been appointed*.

Of course, no model can possibly (or should) capture the entire global state. It is in fact sufficient to keep track of credentials (since they can be requested and sent) and the current role activations (recall that roles can be seen as attributes that can be set and unset).

7.1 Labelled Transition System

We define an operational semantics for the four operations that an entity E_r (the *requester*) can request from an entity E_s (the *service*) acting as a CASSANDRA service: performing an action, activating a role, deactivating a role, and requesting a credential. The state-changing effect of these requests is modelled by a labelled transition system with transitions between global policy sets $\mathcal{P}$, since both credentials and current role activations are captured by policies. The transitions are of the form

$$\mathcal{P} \xrightarrow{\lambda} \mathcal{P}',$$

where the label λ of the transition system consists of four parameters:

- The requester E_r.
- The service E_s to which the request is sent.
- The request, which can be of four different forms:
  1. $\mathsf{doAction}(A)$, where A is an action with ground parameters.
  2. $\mathsf{activate}(R)$, where R is a role with ground parameters.
  3. $\mathsf{deactivate}(E_v, R)$, where the "victim" E_v is an entity and R a role with ground parameters.
  4. $\mathsf{reqCred}(C)$ where C is the requested credential.
- A set C_r of (copy of) credentials submitted to E_s along with the request. We require that C_r is a subset of $\mathcal{P}$, and that all credentials in C_r have the location E_r.

The last condition ensures that the submitted credentials really belong to the requester. The only way to acquire new credentials issued by an entity E_s is via $\mathsf{reqCred}$ requests at E_s's service. To model the submission of credentials, we define a function for renaming the location of a set of credentials:

Definition 7.1.1. *The function $Submit_{E_s}$ takes a set of credentials and renames the location of all credentials of that set to E_s:*

$$Submit_{E_s}(C_r) = \{E_s \Diamond E_{iss}.p \leftarrow c \mid E \Diamond E_{iss}.p \leftarrow c \in C_r\}$$

The following sections define the transition rule for each of the four requests and discuss various alternative designs.

7.2 Action Requests

This transition models a requester E_r performing the action A on a service E_s, and submitting a set C_r of credentials to support the request.

$$\mathcal{P} \xrightarrow{E_r, E_s, \mathsf{doAction}(A), C_r} \mathcal{P} \tag{7.1}$$

The transition has the side condition that

- $\{\mathsf{true}\}$ is an answer to the query $E_s \Diamond E_s.\mathsf{permits}(E_r, A)$, given a global policy set $\mathcal{P} \cup \mathit{Submit}_{E_s}(C_r)$.

Even for this very simple rule, there are alternative definitions. For example, in our version, the service forgets about the submitted credentials. Instead, the rule could specify that the service caches the submitted credentials, so the resulting new policy credential set would be $\mathcal{P} \cup \mathit{Submit}_{E_s}(C_r)$. In effect, such a transition rule would increase the size of E_s's policy by adding to it all credentials ever submitted, and services would be prone to the obvious denial-of-service attack.

Such a caching transition rule would also change the dynamics of the policy in a subtle way: in the original rule, the policy could require the requester E_r to personally submit the credentials; with the new rule, somebody else could also submit them before E_r issues the request. Policy writers must therefore be aware of how exactly the access control engine uses the policy.

7.3 Role Activation

This transition models a requester E_r activating the role R on a service E_s, and submitting a set C_r of credentials to support the request.

$$\mathcal{P} \xrightarrow{E_r, E_s, \mathsf{activate}(R), C_r} \mathcal{P}' \tag{7.2}$$

where $\mathcal{P}' = \mathcal{P} \uplus \{E_s \Diamond E_s.\mathsf{hasActivated}(E_r, R)\}$, so this transition adds a new hasActivated credential rule to E_s's policy. The transition has the following side conditions:

- The role has not already been activated: $\{\mathsf{false}\}$ is an answer for the query $E_s \Diamond E_s.\mathsf{hasActivated}(E_r, R)$.
- $\{\mathsf{true}\}$ is an answer for the query $E_s \Diamond E_s.\mathsf{canActivate}(E_r, R)$, given a global policy set $\mathcal{P} \cup \mathit{Submit}_{E_s}(C_r)$.

According to this rule, role activation fails if the role in question is already activated. Alternatively, a rule could omit this condition and always succeed whenever the corresponding canActivate predicate can be deduced.

7.4 Role Deactivation

This transition models a requester E_r deactivating the "victim" E_v's role R on a service E_s, and submitting a set C_r of credentials to support the request.

$$\mathcal{P} \xrightarrow{E_r, E_s, \mathsf{deactivate}(E_v, R(\vec{e})), C_r} \mathcal{P} - \mathit{Victims}_s \tag{7.3}$$

The transition has the following side conditions:

- E_v has activated $R(\vec{e})$: $\{\mathsf{true}\}$ is an answer for the query $E_s \Diamond E_s.\mathsf{hasActivated}(E_v, R(\vec{e}))$.
- $\{\mathsf{true}\}$ is an answer for the query $E_s \Diamond E_s.\mathsf{canDeactivate}(E_r, E_v, R(\vec{e}))$, given the global policy set $\mathcal{P} \cup \mathit{Submit}_{E_s}(C_r)$.

- The set $\mathit{Victims}_s$ is the set of all hasActivated credential rules in E_s's policy for which a corresponding isDeactivated credential can be derived under the assumption that the predicate $E_s \Diamond E_s.\mathsf{isDeactivated}(E_v, R(\vec{e}))$ holds, or more formally:

$$\begin{aligned}\mathit{Victims}_s = \{ & E_s \Diamond E'_s.\mathsf{hasActivated}(E'_v, R') \in \mathcal{P} \mid \\ & \{\mathsf{true}\} \text{ is an answer to the query } E_s \Diamond E'_s.\mathsf{isDeactivated}(E'_v, R') \\ & \quad \text{under a global policy set } \mathcal{P} \cup \{E_s \Diamond E_s.\mathsf{isDeactivated}(E_v, R)\} \ \}\end{aligned}$$

As a result of the transition, all role activations in $\mathit{Victims}_s$ are removed from E_s's policy. This feature is used to implement cascading revocation. However, note that $\mathit{Victims}_s$ contains only role activations with location E_s; we decided not to support distributed cascading revocation across the network, as is proposed in OASIS [HBM98, YMB02, BMY02]. Such a mechanism would be very hard to model and to implement on a wide scale as it would require an asynchronous event infrastructure [BMB+00] and a much bigger state that records which entities have to be notified about which deactivation events.

The design space of role deactivation allows for many other alternative specifications. In OASIS, role deactivation is triggered when prerequisite predicates that were used to activate the role cease to hold. This semantics can lead to unexpected non-deterministic behaviour if there are several rules specifying different conditions for activating the role: depending on which activation rule is chosen by the system (and the user will generally be unaware of the choice), deactivation may be triggered or not. We avoid this form of non-determinism by separating activation from deactivation rules. A consequence of our approach is the flexibility to specify deactivation conditions that are independent from the conditions used to activate a role. Yet another approach to overcome the non-determinism inherent in OASIS would be to perform the deactivation only if the role cannot be activated by any other means. However, this might be prohibitively expensive in practice.

OASIS also supports automatically triggered deactivation when predicates that depend on the environment or external conditions cease to hold. Again, this feature relies on an event infrastructure and on the particular implementations of such predicates. As specified in the given rule, we only consider deactivations triggered by other deactivations.

There are also several different options concerning who is permitted to deactivate roles. For OASIS, this question is discussed in [BMY02], in the context of role appointment. Three different options are possible in OASIS: with *appointer-only* revocation policies, only the appointer can revoke the appointment certificate. In contrast, *appointer-role* revocation allows anyone who is active in the appointer role to revoke it. Finally, *resignation* revocation refers to the appointee revoking her own appointment. In CASSANDRA, deactivation permissions are specified via canDeactivate rules and are thus much more flexible. They can implement any combination and variants of these "standard" options. Our case study (Chapter 9) has examples where revocation rights are granted to persons who are not even members of the appointer role. These and other examples demonstrate that CASSANDRA's flexibility is needed in real-world applications.

7.5 Credential Requests

This transition models an entity E_r requesting the credential $E_s \Diamond E_{iss}.p(\vec{x}) \leftarrow c$ from a service E_s, and submitting a set C_r of credentials to support the request.

$$\mathcal{P} \xrightarrow{E_r, E_s, \mathsf{reqCred}(E_s \Diamond E_{iss}.p(\vec{x}) \leftarrow c), C_r} \mathcal{P} \cup \mathit{Creds} \tag{7.4}$$

is a valid transition provided the following side conditions hold. Let $\vec{c}_0$ be an answer for the query

$$E_s \Diamond E_s.\mathsf{canReqCred}(E_r, E_{iss}.p(\vec{x})) \leftarrow c$$

under the global policy set $\mathcal{P} \cup \mathit{Submit}_{E_s}(C_r)$. Then we have two cases:

- if $E_s = E_{iss}$, then *Creds* only contains the single credential $E_r \Diamond E_s.p(\vec{x}) \leftarrow \bigvee \vec{d}$ where $\vec{d}$ is a satisfiable answer for the query $E_s \Diamond E_s.p(\vec{x}) \leftarrow \bigvee \vec{c}_0$ under the global policy set $\mathcal{P} \cup \mathit{Submit}_{E_s}(C_r)$.
- Otherwise, $E_s \neq E_{iss}$, in which case *Creds* is the set of all credentials $E_s \Diamond E_{iss}.p(\vec{x}) \leftarrow d \in \mathcal{P}$ such that $d \Rightarrow \bigvee \vec{c}_0$, but, of course, with locations renamed from E_s to E_r.

The initial canReqCred test checks whether the service is willing to disclose a (potentially sensitive) credential of the requested type to the requester. The case split is due to the fact that only if the requested credential is local can they be created and signed freshly. On the other hand, if the requested credential is a foreign credential, the reply will be a *set* of matching credentials that must be collected from the foreign credentials the service already possesses. In both cases, the returned credential(s) can have constraints that are more restrictive than the one originally requested. For example, a reply to a request of a credential

$$\mathtt{UCam} \Diamond \mathtt{UCam}.\mathsf{canActivate}(x, \mathtt{Student}(\mathit{subj})) \leftarrow x = \mathtt{Alice}$$

might be the more restrictive credential

$$\mathtt{UCam} \Diamond \mathtt{UCam}.\mathsf{canActivate}(x, \mathtt{Student}(\mathit{subj})) \leftarrow x = \mathtt{Alice} \wedge \mathit{subj} = \mathtt{Maths}.$$

Again, the design space for credential management is large. A feature that makes CASSANDRA unique in this aspect is its tight integration with automated trust negotiation (ATN) [WSJ00]. Up to now, ATN has always only been considered separately from policy specification. In CASSANDRA, canReqCred rules protecting credentials are specified just like other rules. Together with the reqCred rule and the query evaluation algorithm, they provide ATN "for free".

The paper [GJ00b] discusses the difference between *offline* and *online signing*. Online signing refers to the creation of freshly signed certificates upon a request; this corresponds with the first case in our transition rule, where the requested credential is local. However, issuing credentials online may be considered too expensive, or too insecure as it makes use of the private key. Offline signing attempts to construct a reply from pre-computed certificates only, similar to the second case in our transition rule, where the requested credential is foreign.

Another, more subtle design issue is concerned with the form of the freshly created credential in the case where the requested credential is local. In our current transition rule, only a single credential is created and sent to the requester. In general, the constraint of the credential is a disjunction of smaller constraints. If the cost of online signing is not considered to be too severe, the service could also instead return a set of credentials, each with a different disjunction-less constraint. Such an approach would, of course, place a heavier computational burden on the service, but the set of separate credentials are potentially more useful to the requester: at some point in the future, other entities may request such

a (now foreign) credential from the original requester himself, and the separate credentials can form more accurate answers.

7.6 Scenario Revisited

Consider the example from §3.4 where we had the following four rules:

$$\begin{aligned}&\mathsf{canActivate}(e, \texttt{Admin}()) \leftarrow \\ &\quad \mathsf{hasActivated}(e, \texttt{User}()), \\ &\quad e \in \{\texttt{Alice}, \texttt{Bob}\}\end{aligned} \tag{7.5}$$

$$\mathsf{canDeactivate}(e_1, e_2, \texttt{User}()) \leftarrow e_1 = e_2 \tag{7.6}$$

$$\begin{aligned}&\mathsf{isDeactivated}(e, \texttt{Admin}()) \leftarrow \\ &\quad \mathsf{isDeactivated}(e, \texttt{User}())\end{aligned} \tag{7.7}$$

$$\mathsf{hasActivated}(\texttt{Alice}, \texttt{User}()) \leftarrow \tag{7.8}$$

Let these four rules be denoted by R_1, R_2, R_3 and R_4, respectively, let $E_s = \texttt{Service}$ be the default location and issuer and

$$\mathcal{P}_1 = \{R_1, R_2, R_3, R_4\}.$$

Using the same scenario as in §3.4, we will now illustrate the operational semantics presented in this chapter.

Alice requests to have the Admin() role activated for her. This request is successful, by Transition Rule 7.2, since

$$\mathsf{hasActivated}(\texttt{Alice}, \texttt{Admin}()) \notin \mathcal{P}_1$$

and furthermore, {true} is an answer to the query canActivate(Alice,Admin()) by R_1 and R_4, so we have

$$\mathcal{P}_1 \xrightarrow{\texttt{Alice},\texttt{Service},\texttt{activate}(\texttt{Admin}()),\emptyset} \mathcal{P}_2$$

where $\mathcal{P}_2 = \mathcal{P}_1 \uplus \{\mathsf{hasActivated}(\texttt{Alice}, \texttt{Admin}())\}$.

Now suppose Alice requests that her User() role be deactivated. This request succeeds by Transition Rule 7.3, since true is an answer to the query hasActivated(Alice,User()) according to R_4. Furthermore, true is an answer to the query canDeactivate(Alice,Alice,User()) because of R_2. The transition is then as follows:

$$\mathcal{P}_2 \xrightarrow{\texttt{Alice},\texttt{Service},\texttt{deactivate}(\texttt{Alice},\texttt{User}()),\emptyset} \mathcal{P}_2 - \mathit{Victims}$$

where, by Transition Rule 7.3 and R_3,

$$\begin{aligned}\mathit{Victims} &= \{\mathsf{hasActivated}(V, R(\vec{e})) \in \mathcal{P}_2 \mid \\ &\qquad \{\mathsf{true}\} \text{ is an answer to the query } \mathsf{isDeactivated}(V, R(\vec{e})), \\ &\qquad\quad \text{given the global policy set } \mathcal{P}_2 \cup \{\mathsf{isDeactivated}(\texttt{Alice}, \texttt{User}())\}\ \} \\ &= \{\mathsf{hasActivated}(\texttt{Alice}, \texttt{User}()), \mathsf{hasActivated}(\texttt{Alice}, \texttt{Admin}())\}\end{aligned}$$

8
Policy Idioms

Policy idioms are general procedures and patterns that appear over and over again in policies — not just electronic policies, as the following examples show.

- Role Hierarchy: e.g. "*project leaders are permitted to access anything that product engineers on the same project can access.*"
- Separation of Duties: e.g. "*the initiator of a bank payment must be different from the authoriser.*"
- Cardinality Constraints: e.g. "*any person desirous of becoming a fellow must be recommended [...] by three or more fellows.*"[1]
- Role Delegation: e.g. "*project leaders can temporarily delegate their access privileges to product engineers.*"
- Role Appointment: e.g. in the UK, the Queen appoints the Prime Minister, without holding the privileges and responsibilities of the appointed role herself.

Unlike other policy languages, CASSANDRA does not provide dedicated, special-purpose language constructs for such policy idioms. CASSANDRA was designed to be general and flexible, and can express a whole range of idioms directly, without the need of adding *ad hoc* constructs. This difference in approach is significant: it implies that CASSANDRA, equipped with a sufficiently powerful constraint domain, can also express variants and combinations of standard policy idioms, as well as hitherto unknown ones. Indeed, our work on policies for a nation-wide EHR service has shown that, in large-scale real-world applications, these "standard" policy idioms occur in many variants and combinations with subtle but significant semantic differences (Chapter 9).

This approach not only keeps the language and its semantics small and simple; it also avoids the necessity of having to constantly extend the language. It should be noted that

[1] From the constitution of the Cambridge Philosophical Society.

CASSANDRA was designed specifically for authorisation policies; in particular, we do not consider *obligation policies* specifying the automatic triggering of actions, as in policy systems like Ponder [DDLS01, Dam02].

In the following, we show how standard policies can be written in CASSANDRA.

8.1 Role Validity Periods

In the following rule, a certified doctor (with certification issued at time t) is also member of the role `Doc()` if t is at most one year ago. This is an example where the freshness requirement of a certification is set by the acceptor (as recommended in [Riv98]), not by the certificate issuer. The chosen constraint domain must contain a built-in function that returns the current time and must support integer order constraints.

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{Doc}()) \leftarrow \\ &\quad \mathsf{canActivate}(x, \texttt{CertDoc}(t)), \\ &\quad \texttt{CurTime}() - \texttt{Years}(1) \leq t \leq \texttt{CurTime}()\end{aligned}$$

Note that CASSANDRA cannot express a policy that deactivates the role automatically once the validity period has run out. This would require automated deactivation conditional on environmental predicates, a feature only supported by OASIS, as discussed in §7.4.

8.2 Auxiliary Roles

Sometimes a role is used solely to express some property about its members and can be used without prior activation; such a role is called *auxiliary role*. In this rule, a logged-in user can read a file provided that the system can deduce she is the owner of that file. Ownership is here expressed with the auxiliary `Owner` role that need not be activated.

$$\begin{aligned}&\mathsf{permits}(x, \texttt{Read}(\mathit{file})) \leftarrow \\ &\quad \mathsf{hasActivated}(x, \texttt{Login}()), \\ &\quad \mathsf{canActivate}(x, \texttt{Owner}(\mathit{file}))\end{aligned}$$

8.3 Role Hierarchy

In this variant of parameterised role hierarchy, a project leader is more senior than both a production engineer and a quality engineer. Both production engineer and quality engineer are more senior than the engineer role. This example is taken from [SBC+97]. The hierarchy graph can be directly represented by canActivate dependencies. We extend the example

by using roles that have a "department" parameter:

$$\begin{aligned}
&\mathsf{canActivate}(x, \texttt{Prod-eng}(dep)) \leftarrow \\
&\quad \mathsf{canActivate}(x, \texttt{Proj-leader}(dep)) \\
&\mathsf{canActivate}(x, \texttt{Qual-eng}(dep)) \leftarrow \\
&\quad \mathsf{canActivate}(x, \texttt{Proj-leader}(dep)) \\
&\mathsf{canActivate}(x, \texttt{Eng}(dep)) \leftarrow \\
&\quad \mathsf{canActivate}(x, \texttt{Prod-eng}(dep)) \\
&\mathsf{canActivate}(x, \texttt{Eng}(dep)) \leftarrow \\
&\quad \mathsf{canActivate}(x, \texttt{Qual-eng}(dep))
\end{aligned}$$

8.4 Separation of Duties

To encode separation-of-duties constraints it is necessary to be able to express negated conditions such as "x has *not* activated role $R(y)$", where x and y will have been instantiated by the time the condition is processed. We will write this condition in the body as $\neg\mathsf{hasActivated}(x, R(y))$ as shorthand for the user-defined aggregation condition $\mathsf{existsActivation}_R(0, x, y)$, defined by a rule

$$\begin{aligned}
&\mathsf{existsActivation}_R(\mathsf{count}\langle x'\rangle, x, y) \leftarrow \\
&\quad \mathsf{hasActivated}(x', R(y)), x' = x
\end{aligned}$$

Clearly, $\mathsf{existsActivation}_R(0, x, y)$ holds if and only if x has not activated $R(y)$.

In this common example for separation of duties, a payment transaction requires two phases, initiation and authorisation, which have to be executed by two different people. The rule implements a dynamic and parameterised variant of separation of duties: an `Authoriser` of a payment must not have activated the `Initiator` role for the same payment.

$$\begin{aligned}
&\mathsf{canActivate}(x, \texttt{Authoriser}(payment)) \leftarrow \\
&\quad \neg\mathsf{hasActivated}(x, \texttt{Initiator}(payment))
\end{aligned}$$

As an example for dynamic n-wise parameter-centric separation of duties, suppose that nobody can work on two projects at the same time if they both belong to a set of n pairwise mutually conflicting projects. With a function `Conflict()` that returns this set of conflicting projects, this can be encoded as

$$\begin{aligned}
&\mathsf{canActivate}(x, \texttt{Projmem}(p)) \leftarrow \\
&\quad \mathsf{numberOfActivatedConflictingProjects}(conflicts, x), \\
&\quad conflicts = 0 \\
&\mathsf{numberOfActivatedConflictingProjects}(\mathsf{count}\langle p\rangle, x) \leftarrow \\
&\quad \mathsf{hasActivated}(x, \texttt{Projmem}(p)), \\
&\quad p \in \texttt{Conflict}()
\end{aligned}$$

8.5 Cardinality Constraints

Cardinality constraints are conditions on the number of subjects active in a role. Such conditions can easily be expressed using aggregation rules. Suppose a server only allows up

to 20 users at any given time. The following two rules implement this policy:

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{User}()) \leftarrow \\ &\quad \mathsf{countUsers}(n), \\ &\quad n < 20 \\ &\mathsf{countUsers}(\mathsf{count}\langle x\rangle) \leftarrow \\ &\quad \mathsf{hasActivated}(x, \texttt{User}())\end{aligned}$$

Often it is necessary to express a condition stating that *nobody* has activated a certain role. Such a statement implicitly contains a universally quantified negated predicate and cannot be expressed in logic languages with negated predicates, e.g. Halpern and Weissman's logic [HW03]. With aggregation, this kind of negation can easily be expressed. The following two rules specify that there you can only activate the role `TheOne` if nobody else has activated it.

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{TheOne}()) \leftarrow \\ &\quad \mathsf{countTheOnes}(0) \\ &\mathsf{countTheOnes}(\mathsf{count}\langle x\rangle) \leftarrow \\ &\quad \mathsf{hasActivated}(x, \texttt{TheOne}())\end{aligned}$$

The count aggregation operator can also be used to express *manifold constraints* where n distinct entities together perform some action. Suppose a club requires that a person is eligible to membership if nominated by one proposer and three seconders. We can express this condition as follows:

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{Member}()) \leftarrow \\ &\quad \mathsf{hasActivated}(y, \texttt{Proposer}(x)) \\ &\quad \mathsf{countSeconders}(n, x), \\ &\quad n \geq 3 \\ &\mathsf{countSeconders}(\mathsf{count}\langle y\rangle, x) \leftarrow \\ &\quad \mathsf{hasActivated}(y, \texttt{Seconder}(x))\end{aligned}$$

Now suppose the club also specifies that members can only be seconders for at most five persons at any time. This is an example of a cardinality constraint where we apply the aggregation operator to the role parameter:

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{Seconder}(y)) \leftarrow \\ &\quad \mathsf{canActivate}(x, \texttt{Member}()) \\ &\quad \mathsf{countSeconded}(n, x), \\ &\quad n \leq 5 \\ &\mathsf{countSeconded}(\mathsf{count}\langle x\rangle, y) \leftarrow \\ &\quad \mathsf{hasActivated}(y, \texttt{Seconded}(x))\end{aligned}$$

8.6 Role Delegation

Here, an administrator can delegate her role to somebody else by activating the `DelegateAdm` role for the delegatee. The delegatee can then subsequently activate the administrator role. The first parameter of the administrator role specifies who the delegator was. The second parameter n is an integer for restricting the length of the *delegation chain*

(cf. [ZAC03]): the delegatee can activate the administrator role only with a "rank" n' that is strictly less than the delegator's rank n but must be at least 0. Setting the parameter to 1 for non-delegated administrators (i.e., those at the top of a delegation chain) would implement *non-transitive* delegation. Removing the constraint on n in the second rule would enable *unbounded* delegation chains.

$$\begin{aligned}&\mathsf{canActivate}(x, \texttt{DelegateAdm}(y,n)) \leftarrow \\ &\quad \mathsf{hasActivated}(x, \texttt{Adm}(z,n)) \\ &\mathsf{canActivate}(y, \texttt{Adm}(x,n')) \leftarrow \\ &\quad \mathsf{hasActivated}(x, \texttt{DelegateAdm}(y,n)),\ 0 \leq n' < n\end{aligned}$$

With the following rule, the delegated role is automatically revoked if the delegation role of the delegator is deactivated.

$$\begin{aligned}&\mathsf{isDeactivated}(y, \texttt{Adm}(x,n')) \leftarrow \\ &\quad \mathsf{isDeactivated}(x, \texttt{DelegateAdm}(y,n))\end{aligned}$$

However, we need to specify who is allowed to deactivate a delegation role. In grant-dependent revocation (first rule below), only the delegator herself has this power. In grant-independent revocation (second rule below), every administrator (who has at least as high a rank as the delegator) can deactivate the delegation.

$$\begin{aligned}&\mathsf{canDeactivate}(x, z, \texttt{DelegateAdm}(y,n)) \leftarrow x = z \\ &\mathsf{canDeactivate}(x, z, \texttt{DelegateAdm}(y,n)) \leftarrow \\ &\quad \mathsf{hasActivated}(x, \texttt{Adm}(w,n')),\ n \leq n'\end{aligned}$$

A rather paranoid policy may specify cascading revocation: if a delegated administrator is revoked from her role, all her delegation must also be revoked recursively.

$$\begin{aligned}&\mathsf{isDeactivated}(x, \texttt{DelegateAdm}(y,n)) \leftarrow \\ &\quad \mathsf{isDeactivated}(z, \texttt{DelegateAdm}(x,n'))\end{aligned}$$

8.7 Role Appointment

Delegation can be viewed as a special case of the more general *appointment* mechanism where the appointer is required to be a member of the appointed role [HBM98, YMB02, BMY02]. Appointment can easily be encoded in CASSANDRA. Here we only show the encoding of a simple version of appointment.

A manager M can appoint A as an employee by activating an "appointer role" $\texttt{AppointEmployee}(A)$. This then enables A to activate the "employee appointed by M" role $\texttt{Employee}(M)$.

$$\begin{aligned}&\mathsf{canActivate}(mgr, \texttt{AppointEmployee}(emp)) \leftarrow \\ &\quad \mathsf{hasActivated}(mgr, \texttt{Manager}()) \\ &\mathsf{canActivate}(emp, \texttt{Employee}(appointer)) \leftarrow \\ &\quad \mathsf{hasActivated}(appointer, \texttt{AppointEmployee}(emp))\end{aligned}$$

Furthermore, A's employee role is revoked automatically when $\texttt{AppointEmployee}(A)$ is de-

activated:

$$\mathsf{isDeactivated}(\mathit{emp}, \texttt{Employee}(\mathit{appointer})) \leftarrow$$
$$\quad \mathsf{isDeactivated}(\mathit{appointer}, \texttt{AppointEmployee}(\mathit{emp}))$$

We also have to specify who is allowed to revoke the appointment role. With grant-dependent revocation, only the appointer herself can revoke it:

$$\mathsf{canDeactivate}(x, \mathit{appointer}, \texttt{AppointEmployee}(\mathit{emp})) \leftarrow$$
$$\quad x = \mathit{appointer}$$

Grant-independent revocation, on the other hand, allows every manager to revoke employee roles:

$$\mathsf{canDeactivate}(x, \mathit{appointer}, \texttt{AppointEmployee}(\mathit{emp})) \leftarrow$$
$$\quad \mathsf{hasActivated}(x, \texttt{Manager}())$$

In some cases, all roles appointed by *M* should be revoked whenever *M* is revoked from her role herself, in this case the manager role. We can encode a cascading chain of revocations as follows:

$$\mathsf{isDeactivated}(\mathit{mgr}, \texttt{AppointEmployee}(\mathit{emp})) \leftarrow$$
$$\quad \mathsf{isDeactivated}(\mathit{supermgr}, \texttt{AppointManager}(\mathit{mgr}))$$

8.8 Trust Negotiation and Credential Fetching

Suppose the following rule is part of the policy of a server holding the electronic health records (EHR) for some part of a country's population[2]. To activate the doctor role, x must be a certified doctor in some health organisation *org*, and furthermore the organisation must be a certified health organisation. Both requirements must be satisfied in the form of credentials signed by some entity *auth* belonging to a locally defined set of registration authorities.

$$\mathsf{canActivate}(x, \texttt{Doc}(\mathit{org})) \leftarrow$$
$$\quad \mathit{auth}.\mathsf{canActivate}(x, \texttt{CertDoc}(\mathit{org})),$$
$$\quad \mathit{org} \Diamond \mathit{auth}.\mathsf{canActivate}(\mathit{org}, \texttt{CertHealthOrg}()),$$
$$\quad \mathit{auth} \in \texttt{RegAuthorities}()$$

In the rule above, the location prefix in front of the first body predicate has been omitted, so the doctor certification credential is required to already be in the local policy or be submitted by x together with a role activation request. On the other hand, there is a location prefix *org* in front of the second body predicate: the health organisation credential is automatically requested from *org* over the network, or, more precisely, the entity the variable *org* is instantiated to during actual query evaluation.

However, the health organisation (say, the Elizabeth Hospital) will allow credential disclosure only if its canReqCred policy allows it. With the following rule, the hospital specifies

[2]This example is not from our EHR case study (although it is inspired by it).

that it is willing to reveal its `CertHealthOrg` credential to certified EHR servers.

$$\begin{aligned}&\mathsf{canReqCred}(x, y.\mathsf{canActivate}(z, \texttt{CertHealthOrg}()) \leftarrow\\ &\quad x \Diamond \mathit{auth}.\mathsf{canActivate}(x, \texttt{CertEHRServ}()),\\ &\quad z = \texttt{ElizabethHosp},\\ &\quad \mathit{auth} \in \texttt{RegAuthorities}()\end{aligned}$$

The prefix "$x \Diamond \mathit{auth}$." specifies that the required credential must be signed by some registration authority and that it is to be retrieved automatically from x; in this case, x will have been instantiated to be the EHR server. The EHR server will in turn have canReqCred policy rules specifying to whom its `CertEHRServ` credential may be disclosed. As this example shows, a simple request can trigger multiple phases of credential exchanges between two or more entities over the network until a sufficient level of mutual trust has been established.

9 Case Study: Electronic Health Records

We have described CASSANDRA, a carefully designed trust management system with a flexible and expressive policy language with a formal foundation. But does it fulfil the requirements of real-world applications? And what are actually the requirements of real-world applications? Given the amount of research in the area of trust management in the last few years, there is surprisingly little in the literature on large-scale policy examples and case studies. Most are just academic toy examples for the purpose of illustrating a particular policy system.

The lack of case studies is partly due to the fact that there have been only few applications in the past that really required sophisticated trust management and highly expressive policy languages. It is only now that applications are starting to emerge that require the full power of a system like CASSANDRA. The nation-wide Electronic Health Record service being developed within the NHS National Programme for Information Technology is one of the most challenging such applications. In this chapter, we present our case study on constructing a CASSANDRA policy for such a service.

The history and motivation of the NHS programme are outlined in §9.1. §9.2 describes what makes the programme so challenging and why it is currently in such a troubled state. Our case study is concerned with specifying authorisation to access the Spine, the central part of the programme. The CASSANDRA policy rules for the Spine and related services are presented in detail in §9.4. Finally, we illustrate the policy in a scenario in §9.3. Appendix A collects the policy rules.

9.1 The National Programme

Electronic Health Record (EHR) schemes are now being developed in Europe, the United States, Canada and Australia to provide "cradle-to-grave" summaries of patients' records, linking clinical information across the entire health system [Cor02]. In the UK, the National

Health Service (NHS) has been planning since 1998 to develop a EHR service [Nat98, Dep01a] for England.

These plans culminated in the NHS National Programme for Information Technology (NPfIT) in 2002, conceived in a seminar chaired by Prime Minister Tony Blair and attended by the then Secretary of State for Health, Alan Milburn, as well as other senior members of the Department of Health and industrial representatives. NPfIT is now regarded as the largest, most expensive and one of the most complex IT projects in history [Bre05]. Its procurement cost alone amounts to £6 billion, and it is supposed to manage computerised medical data of the entire English population of 50 million people and connect up a huge number of health care providers.

At the heart of the project lies the Care Record Service (CRS), or the Spine, containing the EHRs of all patients in the country. The second service is Choose-and-Book, an electronic service for booking appointments and clinical referrals. Thirdly, the Electronic Transfer of Prescriptions Service (ETP) will allow prescriptions to be sent electronically to patients' pharmacies. The contracts for suppliers were awarded in 2004, and the project will be deployed in several phases until end of 2010.

9.1.1 Spine Architecture

In the plan, the *Spine* is be the central service that holds the EHRs of all NHS patients. Including deceased users and users who have moved abroad, the total number of records is expected to be in the order of 10^8. It provides online read and write access to the records to authorised users; these will mainly be clinicians and personal users (patients and their agents).

The Spine will be supported by the national *Patient Demographic Service* (PDS). It serves as a single, comprehensive and consistent source of up-to-date demographic patient data (e.g. NHS number, name, address, preferred language). This data is accessed by the Spine and other applications for identifying and authenticating personal users.

The national services are large and have to be able to cope with high loads. By 2010, when NPfIT is expected to be completed, there will be an estimated number of 50 million patients and 300 million GP appointments per year, as well as annually 70 million inpatient episodes and out-patient hospital attendances, and about 30 million other health episodes and encounters (§740, [Nat03]).

An integral goal of the NHS National Programme for IT in the NHS (NPfIT) is the deployment of an infrastructure for identification, registration and authentication of users in a secure, standardised and seamless manner across all national and local applications, based on digital credentials and public key technology. Professional users, i.e. clinical and administrative staff, access EHR data in the Spine based on role credentials, issued by NHS-approved *Registration Authorities* (RAs). RAs will also be responsible for managing clinical workgroup membership. The size of an RA can vary considerably. Most RAs will be local to a single health organisation, but some may be "more nationally based" (§730.24.0, [Nat03]). It is conceivable that large RAs could be located on the NHS cluster level of which there are five in England (covering London; North East, Yorkshire and Humberside; South East and South West; East of England and East Midlands; West Midlands and North West). A typical cluster comprises of up to 2000 General Practices and 100 Acute Trusts and other health organisations. An RA policy should therefore be able to cope with up to 200,000 registered health professionals.

Local applications are expected to make use of and interoperate with the national ser-

vices. In particular, local health organisations (e.g. hospitals, doctors' practices) will gradually move from the traditional paper-based records to electronic databases. The records kept on this level are called *Electronic Patient Records* (EPR); summaries of these are used to populate the Spine. This process may take a long time, and the local procedures differ substantially, so the EHR service cannot be deployed on this level by connecting up all existing EPR systems. Health organisations can be as small as single GP practices but could also be entire NHS trusts with up to 500,000 registered patients.

9.1.2 NPfIT Benefits

The potential benefits of NPfIT are huge — if the project proves to be successful. It has the potential to increase the quality of care: because a patient's EHR will contain information about all health episodes across the NHS, clinicians will have access to more complete data. Information such as allergies, blood type, or current medication is particularly important in accident and emergency cases.

The project may also increase efficiency by reducing waiting times through Choose-and-Book. It is also hoped that lab results, e.g. X-ray images, will be shared online to avoid duplication.

Patients will potentially have more autonomy: they will be able to flexibly book appointments, have access to their own records, and, to some extent, also have the ability to control access to their records.

It is sometimes claimed that EHRs are safer than traditional, paper-based records because they can be encrypted and secured using electronic access control mechanisms. This is highly doubtful for reasons discussed in the following, and indeed, the NHS plans have been subject to a fierce controversial public debate on patient confidentiality (see e.g. [Haw03, Rog03, CS03, Col03a, Pal03, Col03c, Cro03, Fou03, Arn03, Ley04, Col04b, Car05, Mul05, Kei05]).

9.2 NPfIT Challenges and Troubles

Large IT projects have often been over-ambitious and prone to disastrous delays and cost overruns, and have often failed. NPfIT might become another such example. The series of resignations of their senior managers is an indication of management troubles: in December 2003, John Pattison resigned from his post as Senior Responsible Owner [Col03b] and was succeeded by Aidon Halligan, who resigned after just six months in the job, in November 2004 [Col04c][1]. David Kwo, the Implementation Director for the London cluster, resigned in April 2005 [Cla05a], and so did Alan Burns, the Implementation Lead, just one month later [Cla05b].

But there are many more warning signs:

- In October 2004, BT, one of NPfIT's main suppliers, was fined £300,000 for having missed their first deadlines [Arn04].
- In November 2004, the Department of Health released new estimates for the total cost of between £18.6 billion and £31 billion [Col04a].
- In January 2005, the National Audit Office (NAO) warned that the deadlines for the Choose-and-Book service will be missed [Nat05].

[1] The Senior Responsible Owner is the most senior position in UK government IT projects and is someone who is actually supposed to oversee the project over its entire lifetime.

- In February 2005, a Medix survey found that the majority of GPs did not support NPfIT and complained that they had not been consulted [Med05].
- In June 2005, IDX was the first key supplier to be dropped because of missed deadlines [Arn05], after warnings issued by IT director Richard Granger in March.

So, what is going wrong? First of all, mixing politics with IT is always problematic. The first phase of NPfIT was supposed to be deployed by the beginning of 2005. This unrealistically tight deadline was probably motivated by Tony Blair wanting to have presentable results before the general election 2005. Department of Health officials repeatedly ignored warnings from governmental and independent bodies and played down the fact GP support has still not been gained. The project has also been criticised for hiding behind smokescreens. For example, the Output-based Specification (OBS), i.e. the document that also specifies patient confidentiality issues, was kept secret until the Guardian got hold of a copy and leaked it.

Communications about the Programme and with the Health Service have been a problem from the start. Neither doctors nor the public have been been adequately informed and consulted, and one of the main fears is that the system will not sufficiently protect patient confidentiality. According to a survey conducted by the Consumers' Association, 72 percent of the respondents said security and confidentiality are a primary concern [Arn03]. At stake is not just the privacy of sensitive personal information but the entire public health care system, as patients will stop confiding in their GPs if they do not trust the electronic system or do not have sufficient control over the use of their data. Already, public confidence is eroded: with NPfIT, the NHS won a 'Most Heinous Government Organisation' Big Brother Award in 2000 and 'Most Appalling Project' Big Brother Award in 2004 [Ley04].

It is equally important to gain clinician buy-in which will fail if the system is cumbersome to use, if the access restrictions are too strict or the response times too high. Indeed, the clinicians' confidence in the project is already eroded as well. At the annual meeting of the British Medical Association in June 2004, delegates voted for a motion which said, "given the uncertainties and lack of consultation on the Care Records Service [and] until GPs' legitimate concerns are answered, GPs should not engage with the Care Records Service" [Col04b].

9.2.1 Technical Challenges

Of course, building a secure system of such an unprecedented scale is more than challenging. A nation-wide EHR system is intrinsically much riskier than the traditional paper-based records. First of all, the electronic system lacks the social control mechanisms that deter people, and especially insiders, from accessing data without authorisation; and outsiders had to physically break in or bribe someone to get access to all the data. Furthermore, the aggregation of so much data in one central place makes it politically and economically highly valuable and thus more prone to abuse, especially in the light of millions of users with potential access.

The technical challenges concerning *information governance*, i.e. the process of protecting patient-identifiable data, are manifold:

- The proposed Spine is extremely large, holding life-long records of 50 million patients.

- It is widely-distributed: central services such as the Spine or the Personal Demographic Service (PDS) interact with tens of thousands of local clinical information systems, all of which must comply to some global policy.
- The rules governing access to patient-identifiable information are complex and reflect the trade-off between patient confidentiality, usability, and legislative constraints. Neither traditional discretionary nor mandatory access control nor RBAC are sufficiently expressive to capture complex "policy idioms" such as Legitimate Relationships, Sealed Envelopes or consent management.
- The requirements are mandated by laws and regulations such as the Data Protection Act, Mental Health Act, Human Fertilisation and Embryology Act, the Abortion Regulations and the Venereal Diseases Regulations, as well as by official guidelines and ethical positions that are all prone to change. Such changes have to be implemented quickly and consistently across all local and central systems and services.
- Many issues are complex, novel, not well understood, and/or controversial. As an example, the Caldicott committee recommended that access to patient-identifiable information should be on a "strict need-to-know basis" [Nat97]. In contrast, common medical code of ethics and professional practice goes further and requires the patient's consent for accessing personal information [SMW93]. Anderson [And96b] stresses the same point in his security policy commissioned by the British Medical Association, and further demands that patients should automatically receive notifications when their data are accessed. It now seems to be current consensus that patient consent should be the basis for access decisions although it is not yet clear when explicit consent has to be sought and when implicit consent can be assumed. Communicating such issues in plain English alone (be it within NHS consultation groups or between the NHS and their IT suppliers) is problematic: any such description is inherently incomplete and ambiguous.

9.2.2 A Case for Policy Specification

Many of these difficulties could be alleviated by specifying information governance rules in a formal, high-level policy language. Policy specification can be used in the NPfIT project for two purposes. Firstly, it should be used as a *communication aid* for issues on information governance, within the NHS as well as between the NHS and their suppliers, and for specification purposes. Informal descriptions should be supplemented by formal and precise policy rules. NPfIT would benefit from this approach for the following reasons:

- A formal policy is unambiguous, precise and yet concise.
- It is much more concrete and specific than current, natural-language specifications such as the OBS, but abstracts away irrelevant implementation details.
- Policy rules are ideal for presenting alternatives and making clear the (often subtle) differences.
- As formal policies are machine-enforceable, it is easy to build a simulator application for stakeholders (NHS experts, suppliers, clinicians and patient representatives) to "play around with" and thus to explore the consequences of the policy or policy alternatives empirically.

- A formal policy precisely specifies the compliance criteria for suppliers. Furthermore, a policy engine fed with the policy could produce randomised test cases for regression testing of implementations.
- Policy rules are amenable to formal analysis, with which security properties of the policy can be proven mathematically.

Secondly, policy specification and automated policy enforcement by a trust management system should be used to govern access control in the actual implementation. The policy and the policy engine would act as a protective layer between the user interface and the restricted system functions and data. Central services such as the Spine or the PDS as well as local systems would benefit from this approach for the following reasons:

- The approach is consistent with established software engineering and security engineering principles: access control should be independent of system implementation.
- It is by far more maintainable and cost-effective: changes in information governance requirements can be implemented quickly and easily simply by amending the high-level policy. There is no need to change and recompile any source code, and to shut down and restart the service.
- Checking and certifying compliance of systems would become simpler as only the high-level policy has to be checked.

Policy specification would greatly simplify the task of implementing, deploying and maintaining the Spine, but it is no panacea. It is in the nature of the application that not everything can be controlled by automated security mechanisms. Especially clinicians need to be trusted to a considerable degree to act responsibly, so as to not make the system too cumbersome to use in practice. For example, there are several procedures where a clinician can act on behalf of a patient (e.g. give consent), but only after personal consultation with the patient. There are also provisions for clinicians to access data against a patient's wishes, but only in emergency or other exceptional circumstances. Such conditions cannot be checked by the policy, so we have to rely on auditing instead. Of course, auditing is only useful if there is someone who checks the audit trails. For this purpose, each health organisation has senior staff responsible for protecting patient information, called Caldicott Guardians.

9.3 Scenario

The following, rather complex, scenario illustrates some of the more challenging requirements of the Spine policy. We use the role and action names from the CASSANDRA policy presented in §9.4.

Anson Arkwright goes to see Dr Zoe Zimmer, his family's General Practitioner (GP), for an HIV test. Dr Zimmer records the visit in a local EPR item[2] by performing an `Add-record-item` action (with suitable parameters) but does not submit a summary to the Spine on Anson's request. Some time later, Bob Arkwright, Anson's father, visits Dr Zimmer because of heart problems. During the visit he also tells her that he believes his son Anson may be a hypochondriac. Dr Zimmer adds a record item to Bob's EPR about his heart condition and an item in Anson's EPR about his father's comments. The latter item

[2]Recall that *EPRs* are the detailed patient records held in health organisations and are meant to replace the traditional paper-based records, whereas *EHRs* are the shared summary records held on the Spine.

is marked as containing third party information about his father, so as long as his father (or the Caldicott Guardian on his behalf) does not enter a `Third-party-consent` role for that item, Anson will not be able to read it. (Note that we use roles not just to model job positions within an organisation but also to indicate state changes, e.g. giving third-party consent.)

Dr Zimmer also attempts to submit a summary of Bob's new EPR item to his shared EHR: she first activates her `Spine-clinician` role on the Spine by submitting an RA-issued `NHS-clinician-cert` role credential along with the activation request. Subsequently, her `Add-spine-record-item` action succeeds because the Spine can deduce she is Bob's `Treating-clinician` (Bob has explicitly consented to treatment years ago and has not withdrawn his `Consent-to-treatment` role). Dr Zimmer also decides to refer Bob to a local hospital's cardiologist, Dr Hannah Hassan. As Bob's treating clinician, Dr Zimmer can enter a `Referrer` role on the Spine, thus enabling Dr Hassan to also become a treating clinician with a *legitimate relationship*. Bob's consent is not needed, but he has the power to cancel the referral by deactivating Dr Zimmer's `Referrer` role.

At the hospital, a `Receptionist` registers Bob as a patient by activating a `Register-patient` role. After his out-patient visit with Dr Hassan, the receptionist registers him with a surgical team in the same hospital for a heart bypass operation. For this purpose, the receptionist activates appropriate `Register-team-episode` and `Register-ward-episode` roles on the hospital's service, thereby establishing a legitimate treating clinician relationship between Bob and the surgical team and the ward nurses. During surgery, abnormal liver values are found, so the team attempts to search for potentially important information in Bob's EHR on the Spine. However, years ago, Bob activated a `Conceal-request` role on the Spine to conceal the contents of all items in his EHR concerning an alcohol-related liver problem from everybody except clinicians treating him as GP, and this request had been granted by Dr Zimmer, who activated a corresponding `Concealed-by-spine-patient` role for this purpose. The head of the surgical team, Dr Lily Littlewood, decides to "break the seal" to view Bob's restricted EHR item by performing the action `Force-read-spine-item`. This can be done by any clinician with a legitimate relationship but will be marked in the audit trail to be investigated by the hospital's Caldicott Guardian.

Unfortunately, the team encounters further complications during the operation and Bob needs to be kept in an artificial coma. Dr Zimmer agrees to appoint Bob's wife, Carol, to be his agent by activating a `Register-agent` role on the Spine. Carol then requests to activate the `Agent` role on the hospital's service. This succeeds after a cycle of trust negotiation between the hospital, the Spine, and the hospital's RA: the hospital's service reacts to Carol's request by requesting an agent registration credential from the Spine; the Spine replies by requesting a health organisation credential; the hospital agrees by sending an health organisation credential, issued by some RA, to the Spine; the Spine requests an NHS-signed credential from the RA to check if it is an officially approved RA, and finally, the Spine sends the originally requested `Register-agent` credential to the hospital certifying that Carol is indeed Bob's agent.

When Bob is woken and released, he attempts to revoke the agent registration for his wife but fails because it was Dr Zimmer who registered Carol. However, on Bob's request, Dr Zimmer deactivates her `Register-agent` role for Carol. If Carol is active with an `Agent` role on the Spine at that moment, cascading revocation causes that role to be deactivated immediately as well.

9.4 EHR Policy

9.4.1 Overview

We have drafted a complete CASSANDRA policy for the NHS Spine and related services, based mainly on the Output Based Specification Version 2.0 (OBS) [Nat03], reports from NHS pilot projects of the Electronic Records Development and Implementation Programme (ERDIP) [Nat02, Gau03], and various Department of Health documents [Dep01b, Dep02].

The OBS is a 900-page document given to potential suppliers during the procurement process in August 2003. According to the OBS, the modules are to be delivered by contractors in three phases. By December 2004 (Phase 1), a basic system for accessing EHRs on the Spine should have been delivered, and the Spine populated with patient data[3]. At this stage, the required confidentiality requirements are dangerously low and have prompted harsh criticism from doctors, patient interest groups and the media: a one-off general consent given by a patient would make his personal data available to all clinical users of the Spine. Stricter confidentiality measures are introduced with the later phases. In Phase 2 (December 2006), patients will directly access their health data, they can request to conceal parts of their records, and can identify people (agents) to act on their behalf concerning access to records. Furthermore, access control based on clinical workgroups will be supported. In Phase 3 (December 2010), clinicians will also be able to conceal parts of their patients' records, access will be based on legitimate relationships between patients and clinicians, and systems must separately deal with data containing confidential third-party information. Our proposed policy covers all requirements regarding the access of patient-identifiable data up to and including Phase 3.

The most relevant section for our case study is section §730 on information governance, a list of security and confidentiality requirements for systems handling patient data. It was specifically written to comply with relevant legislation and guidelines, in particular with the Data Protection Act 1998 and the Caldicott guidelines [Nat97], and, as far as patient registration is concerned, with the 'Registration and Authentication e-Government Strategy Framework Policy and Guidelines'. The document acknowledges that the requirements of this section are likely to evolve due to changes in national guidelines and legal requirements, but also to new positions emerging from still ongoing NHS consultation processes. Some of the points will be affected by design work yet to be done, and some are "subject to further guidance".

Consequently, the requirements are sketchy in places. For example, section §730.9 states the requirements of role-based access control without specifying which roles will be used and their associated privileges. Similarly, it is a Phase 3 requirement that only clinicians with a legitimate relationship have access to patient data (§730.17). However, the section does not explain the rules for establishing legitimate relationships. In writing a formal policy for the system, many such missing details had to be filled in. Many of our rules are based simply on common sense, even though they are not explicitly required in the OBS: for example, that a legitimate relationship is automatically revoked if the respective patient is no longer registered in the system.

In some cases we had to choose from a number of conceivable options and, in general, favoured the more demanding solutions. For example, patients may seal off groups of related clinical data (e.g. all the data about a particular event) (§730.48.2), but the OBS

[3] The deadline was rather optimistic. Not too surprisingly, their main contractor, BT Synergy, failed to deliver the Spine on time and was fined by the NHS.

defers the specification of the granularity of such groupings until further guidance has been produced. Our solution gives patients an extremely high flexibility in specifying sealing-off criteria and may well be more flexible than what is actually needed, but it shows that less demanding solutions could also be implemented in CASSANDRA.

In our case study, the *entities* are the individual users (patients, clinicians, staff) as well as the national and local services. We have written policies for the Spine, the PDS, Addenbrooke's Hospital (as an example for a local health organisation), and Addenbrooke's RA. The policies use the constraint domain $\mathcal{C}_0$(§5.2.4) and comprise 375 rules, 71 roles and 12 actions. Of the 375 rules, there are 118 canActivate, 97 canDeactivate, 51 isDeactivated, 29 permits and 27 canReqCred rules. The remaining 53 are user-defined rules. The case study suggests that common policy idioms such as appointment hardly occur in their pure forms in practice. It is therefore not sufficient to equip a policy language with standard policy constructs (e.g. appointment in OASIS [HBM98, YMB02, BMY02]); rather, it is necessary to be able to express different variants of them. The rules can be roughly divided into the following categories:

Permissions Assignment

Many of the permits rules are straightforward parameterised role-action assignments, e.g. *"patients can annotate their own record items"*. Others require more than one role-related prerequisite condition, e.g. *"clinicians can force-read record items concealed by a patient if they have activated their clinician role and if they are member of a workgroup (clinical team or ward) currently treating the patient"*. The last condition is also an example of an auxiliary or derived role: the `Group-treating-clinician` role need not be activated when using the rule; it is sufficient that it *can* be activated.

The permits rules concerning reading record items are typically also conditional on patient and third-party consent and (absence of) access restrictions. All these conditions correspond to role activations of users other than the requester. Such rules cannot be easily expressed in languages in which the subject parameters of the head and the body are implicitly the same, e.g. SPKI/SDSI [Ell99, EFL+99], RT [LMW02] or OASIS [HBM98, YMB02, BMY02].

Consent

Access to health records is primarily based on explicit patient consent. Consent may be required for initial treatment, for referrals and for disclosure of third-party information. We implement consent as a form of appointment: by activating a consent role, a patient "appoints" a clinician to be e.g. a `Treating-clinician` with a legitimate relationship. To prevent frivolous users from unsolicitedly activating myriads of consent roles, the user must first have been requested to activate the consent role. These consent requests are again implemented as a form of appointment, but now the other way round: by activating a consent request role, the clinician enables the patient to activate a consent role. Consent is thus implemented as a two-stage appointment mechanism.

Registration

Registration is an administrative task that takes on many forms in our case study: for example, PDS managers enter newly born patients into the PDS, receptionists register new patients, human resource managers employ clinicians and other staff, head nurses assign

nurses to wards, and heads of clinical teams assign clinicians to their respective teams. Registration can again be implemented using variants of the appointment encoding given in Chapter 8. Variants include combinations with cardinality restrictions (*"patients can register at most three distinct agents acting on their behalf"*) and uniqueness constraints (*"a patient can only be registered if no one has already activated the registration role for that patient"*). The two mentioned variants make use of CASSANDRA's aggregation operators.

Referral

Referral is implemented as a form of delegation. Our case study exhibits two kinds of patient referral. On the Spine, no patient consent is required, and referral chains are of unbounded length. On the level of the local health organisation, we decided to implement a stricter alternative: a local treating clinician can refer the patient to an external clinician (who will then have restricted rights to read the local EPR record items) only with explicit patient consent, and the delegation chain can only be of length one.

Sealing Off Data

This is a policy idiom motivated by the requirement that patients may specify access restrictions on their data. Patients can fine-tune the access rights to their records by activating an appropriate concealment request role, if the request is subsequently approved by a clinician. The permits rules governing read access need to check that no such concealment role has been activated and approved; this requires universally quantified negation, expressed with the help of aggregation operators.

Deactivation

canDeactivate rules specify who can deactivate which roles. Although these rules are rather straightforward, it is important that deactivation can be specified flexibly. For example, revocation of agent registrations is asymmetric in the sense that patients can only revoke the agents they have appointed themselves (grant-dependent revocation), whereas Caldicott Guardians can revoke not only the agents they have appointed for a patient but also those appointed by the patient (a variant of grant-independent revocation). Furthermore, agent role activations are revoked only if all their registrations have been revoked. This is yet another example of universally quantified negation requiring aggregation operators.

Cascading deactivation, specified by isDeactivated rules, is used to automatically deactivate a role if some other role is deactivated. For example, the revocation of a patient's registration in the hospital triggers the deactivation of all roles that have something to do with that patient, including agent registrations, inpatient episode registrations, legitimate relationships with clinicians, access restriction roles, and consent roles.

Credential Management

Credential-based trust negotiation and credential protection are governed by the interaction between canReqCred rules and rules with remote body predicates. The scenario in §9.3 gives an example of multi-phase automatic trust negotiation. canReqCred rules are also used for regulating direct credential requests from entities. For example, agent credentials from the Spine can be requested by certified health organisations, and also by the agent himself. The location parameter of CASSANDRA predicates facilitates very flexible forms of automated

credential retrieval: unlike other policy systems, credential locations are not restricted to the issuer or the credential subject. For example, a credential of the form

$$\texttt{RA}.\mathsf{hasActivated}(\texttt{RA-adm}, \texttt{NHS-health-org-cert}(\texttt{Org}, \texttt{Start}, \texttt{End}))$$

may be found at the location `Org` which is neither the issuer (`RA`) nor the subject (`RA-adm`).

In the following we describe in detail the policies for the EHR architecture outlined in §9.1.1 and illustrated in the scenario (§9.3). All rules are also listed in the Appendix A. The rules have been parsed and automatically typeset by our prototype implementation.

9.4.2 The Spine

The Spine is the heart of NPfIT, containing EHRs for all patients and providing online access to the records to both to both patients and professional users. Our policy for the Spine defines the access roles and privileges, and manages consent, legitimate relationships, and concealment of record items. The Spine policy comprises 137 rules.

EHR Structure

Each patient is associated with exactly one EHR, an append-only structure consisting of a set of immutable record items, indexed by an ID. We thus assume that each record item is uniquely identified by a pair (*pat*, *id*): the item with ID *id* in the EHR of patient *pat*. The policy accesses relevant fields of a record item via the following system functions, each of which take such a pair as argument:

- `Get-spine-record-author` returns the author of the item. This is always a clinician.
- `Get-spine-record-org` returns the health organisation of the author (at the time of item creation)[4].
- `Get-spine-record-time` returns the time and date of the item's creation.
- `Get-spine-record-subjects` returns a set of subject matters (chosen from a predefined list of valid subjects such as "allergy", "abortion", "cancer") the item relates to.
- `Get-spine-record-third-parties` returns a set of all third parties whose consent must be sought prior to revealing the item to the patient.

As is the case throughout this case study, fields and parameters that are irrelevant to the policy are omitted. For example, there is no "content" field for record items, as the policy does not perform any computations on the actual content of the item. Similarly, alert-raising actions such as `Force-read-spine-record-item`, i.e. the forced reading of an item against the patient's wishes, would in reality contain a "reason" parameter explaining why this action is performed, but it is omitted since it is irrelevant for the policy.

[4] A clinician may be working for several health organisations, but the policy allows her to log on with only one role specifying just one health organisation at any time.

Main Roles

The Spine policy is role-based (§730.9)[5], and designed in such a way that users need to have activated a single main role, `Spine-clinician`, `Spine-admin`, `Patient`, `Agent` and `Third-party`, before performing any action or activating a registration, consent or concealment roles. The requirement that *exactly one* role must be active (§730.12.10) is an example of dynamic n-wise separation of duties (cf. §8.4). Depending on which main role is activated, a different set of permitted actions is presented to the user. The purpose is to restrict the privileges to those needed for the current task (principle of least privilege) and also to simplify auditing.

The activation rules for all these roles contain a user-defined predicate no-main-role-active($user$) (S1.5.3) that is satisfied only if the user has not already activated any of these roles:

(S1.5.3)
no-main-role-active($user$) ←
count-agent-activations(n, $user$),
count-spine-clinician-activations(n, $user$),
count-spine-admin-activations(n, $user$),
count-patient-activations(n, $user$),
count-third-party-activations(n, $user$),
$n = 0$

As in §8.4, this is achieved by the use of aggregation predicates (S1.1.4, S1.2.4, S1.3.4, S1.4.5, S2.2.13).

(S1.1.4)
count-spine-clinician-activations(count⟨u⟩, $user$) ←
hasActivated($user$, `Spine-clinician`(ra, org, $spcty$))

(S1.2.4)
count-spine-admin-activations(count⟨u⟩, $user$) ←
hasActivated($user$, `Spine-admin`())

(S1.3.4)
count-patient-activations(count⟨u⟩, $user$) ←
hasActivated($user$, `Patient`())

(S1.4.5)
count-agent-activations(count⟨u⟩, $user$) ←
hasActivated($user$, `Agent`(pat))

(S2.2.13)
count-third-party-activations(count⟨u⟩, $user$) ←
hasActivated($user$, `Third-party`())

The following describes the rules concerning the main access roles apart from `Third-party` which is discussed in §9.4.2.

[5]Here and in the following, the section numbers starting with 730 relate to the OBS [Nat03].

Clinicians A clinician certified by an NHS-approved RA *ra* working for health organisation *org* with specialty *spcty* can activate the role `Spine-clinician`(*ra*, *org*, *spcty*). The credential can be either submitted locally (S1.1.1) or retrieved from the RA (S1.1.2):

(S1.1.1)
canActivate(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)) ←
    ra.hasActivated(*x*, `NHS-clinician-cert`(*org*, *cli*, *spcty*, *start*, *end*)),
    canActivate(*ra*, `Registration-authority`()),
    no-main-role-active(*cli*),
    `Current-time`() ∈ [*start*, *end*]

(S1.1.2)
canActivate(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)) ←
    ra◇*ra*.hasActivated(*x*, `NHS-clinician-cert`(*org*, *cli*, *spcty*, *start*, *end*)),
    canActivate(*ra*, `Registration-authority`()),
    no-main-role-active(*cli*),
    `Current-time`() ∈ [*start*, *end*]

In both cases, the credential must be still valid, and it is checked if *ra* is an NHS-approved RA (S1.5.1), if necessary, by contacting the RA itself (S1.5.2):

(S1.5.1)
canActivate(*ra*, `Registration-authority`()) ←
    `NHS`.hasActivated(*x*, `NHS-registration-authority`(*ra*, *start*, *end*)),
    `Current-time`() ∈ [*start*, *end*]

(S1.5.2)
canActivate(*ra*, `Registration-authority`()) ←
    ra◇`NHS`.hasActivated(*x*, `NHS-registration-authority`(*ra*, *start*, *end*)),
    `Current-time`() ∈ [*start*, *end*]

A clinician can deactivate her own role (S1.1.3):

(S1.1.3)
canDeactivate(*cli*, *cli*, `Spine-clinician`(*ra*, *org*, *spcty*)) ←

Administrators Administrators are responsible for registering new patients and unregistering deceased patients or those who have moved away. The administrator role `Spine-admin`() can be activated if the person has been registered as such (S1.2.1):

(S1.2.1)
canActivate(*adm*, `Spine-admin`()) ←
    hasActivated(*x*, `Register-spine-admin`(*adm*)),
    no-main-role-active(*adm*)

Administrators can be registered if they have not already been registered (S1.2.5, S1.2.7) and unregistered (S1.2.6) by other administrators:

(S1.2.5)
canActivate(adm, `Register-spine-admin`($adm2$)) ←
    hasActivated(adm, `Spine-admin`()),
    spine-admin-regs($0, adm2$)

(S1.2.7)
spine-admin-regs(count$\langle x \rangle, adm$) ←
    hasActivated(x, `Register-spine-admin`(adm))

(S1.2.6)
canDeactivate(adm, x, `Register-spine-admin`($adm2$)) ←
    hasActivated(adm, `Spine-admin`())

The role can be deactivated by the administrator herself (S1.2.2) and is automatically deactivated when the registration is cancelled (S1.2.3):

(S1.2.2)
canDeactivate(adm, adm, `Spine-admin`()) ←

(S1.2.3)
isDeactivated(adm, `Spine-admin`()) ←
    isDeactivated(x, `Register-spine-admin`(adm))

Patients A patient can activate (S1.3.1) the role `Patient`() if he has been registered on the Spine and also at the PDS. The latter condition is checked by contacting the PDS directly, as required by §730.24.0b:

(S1.3.1)
canActivate(pat, `Patient`()) ←
    hasActivated(x, `Register-patient`(pat)),
    no-main-role-active(pat),
    `PDS`◊`PDS`.hasActivated(y, `Register-patient`(pat))

Patients can be registered if they have not already been registered (S1.3.5, S1.3.7) and unregistered (S1.3.6) by administrators:

(S1.3.5)
canActivate(adm, `Register-patient`(pat)) ←
    hasActivated(adm, `Spine-admin`()),
    patient-regs($0, pat$)

(S1.3.7)
patient-regs(count$\langle x \rangle, pat$) ←
    hasActivated(x, `Register-patient`(pat))

In our policy, the removal of a patient's registration should only be performed if his data is permanently removed from the Spine, for example, in the case of the patient's death, after the legal minimal retention period.

The patient role can be deactivated by the patient himself (S1.3.2) and is automatically deactivated when the registration is cancelled (S1.3.3):

(S1.3.2)
canDeactivate(pat, pat, `Patient`()) ←

(S1.3.3)
isDeactivated(pat, `Patient`()) ←
    isDeactivated(x, `Register-patient`(pat))

Note that the patient role need not be activated in order to get registered as a patient or to receive treatment. Its main purpose is to allow patients access their own data via the NHS Web portal.

Agents A patient can identify agents (for example carers, family members, guardians of a child) who have a special kind of legitimate relationship that allow them to act on the patient's behalf and to access his record (§730.20.10, §730.52). A person can activate (S1.4.1) the role `Agent`(pat) for a patient pat if he has been appointed as an agent and if he is registered at the PDS. As in the case of patient role activation, the latter condition is checked by contacting the PDS directly (P2.2.5):

(S1.4.1)
canActivate(ag, `Agent`(pat)) ←
    hasActivated(x, `Register-agent`(ag, pat)),
    `PDS`◇`PDS`.hasActivated(y, `Register-patient`(ag)),
    no-main-role-active(ag)

(P2.2.5)
canReqCred(org, `PDS`.hasActivated(x, `Register-patient`(pat))) ←
    org◇ra.hasActivated(x, `NHS-health-org-cert`(org, $start$, end)),
    canActivate(ra, `Registration-authority`())

The role can be deactivated by the agent himself (S1.4.2) and is deactivated automatically if all his agent registrations for the patient have been cancelled (S1.4.3, S1.4.4):

(S1.4.2)
canDeactivate(ag, ag, `Agent`(pat)) ←

(S1.4.3)
isDeactivated(ag, `Agent`(pat)) ←
    isDeactivated(x, `Register-agent`(ag, pat)),
    other-agent-regs(0, x, ag, pat)

(S1.4.4)
other-agent-regs(count⟨y⟩, x, ag, pat) ←
    hasActivated(y, `Register-agent`(ag, pat)),
    $x \neq y$

Patients can appoint up to three personal agents (S1.4.9, S1.4.14). This is one of the rules that we invented to prevent a simple kind of denial-of-service attack. To which extent such mechanisms should be enforced by the policy is arguable. We decided to write the policy

in such a way that it generally prevents personal users (patients), but not professional users (clinicians), from frivolously or maliciously clogging up the policy by activating a large number of rules. However, not all denial-of-service attacks can be prevented by the policy itself, e.g. flooding the service with a large number of requests, or submitting a large number of credentials.

(S1.4.9)
$\mathsf{canActivate}(pat, \texttt{Register-agent}(agent, pat)) \leftarrow$
$\quad \mathsf{hasActivated}(pat, \texttt{Patient}()),$
$\quad \mathsf{agent\text{-}regs}(n, pat),$
$\quad n < 3$

(S1.4.14)
$\mathsf{agent\text{-}regs}(\mathsf{count}\langle x \rangle, pat) \leftarrow$
$\quad \mathsf{hasActivated}(pat, \texttt{Register-agent}(x, pat))$

Agents can also be appointed by the patient's GP (S1.4.10). This can be done without the patient's consent if he lacks competence (§730.55.6), for example if the patient is a child and not Gillick-competent[6] but objects to his parents acting on his behalf:

(S1.4.10)
$\mathsf{canActivate}(cli, \texttt{Register-agent}(agent, pat)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{General-practitioner}(pat))$

The patient can only cancel the agent appointments he has made himself but not those made by his GP (S1.4.11) (grant-dependent revocation). A patient's GP, on the other hand, can cancel any agent appointments (S1.4.12) (grant-independent revocation):

(S1.4.11)
$\mathsf{canDeactivate}(pat, pat, \texttt{Register-agent}(agent, pat)) \leftarrow$
$\quad \mathsf{hasActivated}(pat, \texttt{Patient}())$

(S1.4.12)
$\mathsf{canDeactivate}(cli, x, \texttt{Register-agent}(agent, pat)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{General-practitioner}(pat))$

All agent appointments are automatically cancelled if the patient's registration is cancelled (S1.4.13):

(S1.4.13)
$\mathsf{isDeactivated}(x, \texttt{Register-agent}(agent, pat)) \leftarrow$
$\quad \mathsf{isDeactivated}(y, \texttt{Register-patient}(pat))$

[6] A child is *Gillick-competent* if it is deemed mature enough to give or refuse consent to a medical procedure by him or herself. Parental consent is not legally required, only recommended. This was established in the 1985 ruling of the House of Lords in the case Gillick v West Norfolk and Wisbech Area Health Authority.

An agent can request a credential certifying his role as agent (S1.4.6):

(S1.4.6)
canReqCred(ag, Spine.canActivate(ag, Agent(pat))) ←
    hasActivated(ag, Agent(pat))

Such a credential could for example be used for authorisation in the EPR systems of local health organisations, as in the following two rules located at Addenbrooke's Hospital's policy (A1.6.2, A1.6.3):

(A1.6.2)
canActivate($agent$, Agent(pat)) ←
    canActivate(pat, Patient()),
    no-main-role-active($agent$),
    PDS◊PDS.hasActivated(x, Register-patient($agent$)),
    Spine◊Spine.canActivate($agent$, Agent(pat))

(A1.6.3)
isDeactivated(ag, Agent(pat)) ←
    isDeactivated(x, Register-agent(ag, pat)),
    other-agent-regs($0, x, ag, pat$)

Health organisations can also directly ask for such a credential. For this purpose they first have to authenticate themselves with a currently valid RA-signed health organisation credential (S1.4.7). If no such credential is submitted, the Spine tries to retrieve it from the health organisation (S1.4.8). In both cases it is also checked whether the RA is approved by the NHS:

(S1.4.7)
canReqCred(org, Spine.canActivate(ag, Agent(pat))) ←
    ra.hasActivated(x, NHS-health-org-cert(org, $start$, end)),
    canActivate(ra, Registration-authority()),
    Current-time() ∈ [$start$, end]

(S1.4.8)
canReqCred(org, Spine.canActivate(ag, Agent(pat))) ←
    org◊ra.hasActivated(x, NHS-health-org-cert(org, $start$, end)),
    canActivate(ra, Registration-authority()),
    Current-time() ∈ [$start$, end]

Express Consent

It is generally agreed that disclosure of patient data must be preceded by the patient's express consent. There is of course much controversy about the details: how specific should the statement of consent be? How often should consent be sought? When can (implied) consent be derived from express consent? Very strict consent requirement may be good in terms of medical ethics but may be too impractical for users.

The following describes the rules concerning patient consent based on our interpretation of the OBS. This part of the policy is very likely to change as the debate on consent evolves and more guidelines are produced.

One-Off Consent According to §730.4.2, patients will not be allowed to refuse having their medical data stored on the Spine, but the patient's "one-off consent" is required to release the data for clinical care[7]. It should be noted that refusal to give one-off consent will not result in the patient not receiving any clinical treatment. It will only result in clinicians treating the patients not being able to share the data online and having to rely on local patient records.

In the Spine policy, a patient *pat* can give a one-off consent to have his data made available on the Spine by activating the role `One-off-consent`(*pat*) (S2.1.1):

(S2.1.1)
canActivate(*pat*, `One-off-consent`(*pat*)) ←
    hasActivated(*pat*, `Patient`())

Furthermore, his agent can also do this on his behalf (S2.1.2), and so can any treating clinician (i.e. a clinician with a legitimate relationship) (S2.1.3).

(S2.1.2)
canActivate(*ag*, `One-off-consent`(*pat*)) ←
    hasActivated(*ag*, `Agent`(*pat*))

(S2.1.3)
canActivate(*cli*, `One-off-consent`(*pat*)) ←
    hasActivated(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)),
    canActivate(*cli*, `Treating-clinician`(*pat*, *org*, *spcty*))

The patient (S2.1.4), or his agent (S2.1.5) or a treating clinician (S2.1.6) on his behalf, can withdraw the consent by deactivating the role:

(S2.1.4)
canDeactivate(*pat*, *x*, `One-off-consent`(*pat*)) ←
    hasActivated(*pat*, `Patient`())

(S2.1.5)
canDeactivate(*ag*, *x*, `One-off-consent`(*pat*)) ←
    hasActivated(*ag*, `Agent`(*pat*))

(S2.1.6)
canDeactivate(*cli*, *x*, `One-off-consent`(*pat*)) ←
    hasActivated(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)),
    canActivate(*cli*, `Treating-clinician`(*pat*, *org*, *spcty*))

The consent role is automatically deactivated if the patient's registration is cancelled (S2.1.7):

(S2.1.7)
isDeactivated(*x*, `One-off-consent`(*pat*)) ←
    isDeactivated(*y*, `Register-patient`(*pat*))

[7] This is one of the more controversial points as the specifications also allow clinicians to access a patient's records without his consent in "exceptional circumstances" (§730.4.11)

Third-Party Consent Health record items may sometimes contain information about someone other than the patient. A GP may for example include information about certain diseases of the patient's blood relatives [SMW93]. According to the UK Data Protection Act 1998, patients may not view record items that may reveal confidential information about third parties without the third parties' consent. The OBS only requires that any third-party information be withheld from the patient. Our policy also allows patients to request a third party to give their consent.

Third-party consent from a third party x for a particular record item id of a patient pat can be requested by the patient himself (S2.2.1), by his agent (S2.2.2) or by any clinician currently treating him (S2.2.3) by the activation of `Request-third-party-consent`(x, pat, id). A further condition for this request is that x is actually recorded as a third party in the record item (pat, id):

(S2.2.1)
$$\mathsf{canActivate}(pat, \texttt{Request-third-party-consent}(x, pat, id)) \leftarrow$$
$$\quad \mathsf{hasActivated}(pat, \texttt{Patient}()),$$
$$\quad x \in \texttt{Get-spine-record-third-parties}(pat, id)$$

(S2.2.2)
$$\mathsf{canActivate}(ag, \texttt{Request-third-party-consent}(x, pat, id)) \leftarrow$$
$$\quad \mathsf{hasActivated}(ag, \texttt{Agent}(pat)),$$
$$\quad x \in \texttt{Get-spine-record-third-parties}(pat, id)$$

(S2.2.3)
$$\mathsf{canActivate}(cli, \texttt{Request-third-party-consent}(x, pat, id)) \leftarrow$$
$$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty)),$$
$$\quad \mathsf{canActivate}(cli, \texttt{Treating-clinician}(pat, org, spcty)),$$
$$\quad x \in \texttt{Get-spine-record-third-parties}(pat, id)$$

The request can be cancelled by the same users who can activate it (S2.2.4-6). Additionally, the respective third party may also deactivate the request role (S2.2.7), thereby withholding (or later withdrawing) consent to disclosure:

(S2.2.4)
$$\mathsf{canDeactivate}(pat, y, \texttt{Request-third-party-consent}(x, pat, id)) \leftarrow$$
$$\quad \mathsf{hasActivated}(pat, \texttt{Patient}())$$

(S2.2.5)
$$\mathsf{canDeactivate}(ag, y, \texttt{Request-third-party-consent}(x, pat, id)) \leftarrow$$
$$\quad \mathsf{hasActivated}(pat, \texttt{Agent}(pat))$$

(S2.2.6)
$$\mathsf{canDeactivate}(cli, y, \texttt{Request-third-party-consent}(x, pat, id)) \leftarrow$$
$$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty))$$

(S2.2.7)
$$\mathsf{canDeactivate}(x, y, \texttt{Request-third-party-consent}(x, pat, id)) \leftarrow$$
$$\quad \mathsf{hasActivated}(x, \texttt{Third-party}())$$

All requests are automatically deactivated when the patient's registration is cancelled

(S2.2.8):

(S2.2.8)
isDeactivated(x, `Request-third-party-consent`(y, pat, id)) ←
    isDeactivated(z, `Register-patient`(pat))

In order to give consent to third-party disclosure, the third party first has to activate the `Third-party()` role (S2.2.10). This is allowed if his consent has been requested and he is a registered user at the PDS:

(S2.2.10)
canActivate(x, `Third-party()`) ←
    hasActivated(y, `Request-third-party-consent`(x, pat, id)),
    no-main-role-active(x),
    `PDS`◊`PDS`.hasActivated(z, `Register-patient`(x))

The role can be deactivated by the third party (S2.2.11) and is automatically deactivated when all relevant requests have been withdrawn (S2.2.12, S2.2.9):

(S2.2.11)
canDeactivate(x, x, `Third-party()`) ←

(S2.2.12)
isDeactivated(x, `Third-party()`) ←
    isDeactivated(y, `Request-third-party-consent`(x, pat, id)),
    other-third-party-consent-requests($0, y, x$)

(S2.2.9)
other-third-party-consent-requests(count$\langle x \rangle, y, z$) ←
    hasActivated(x, `Request-third-party-consent`(z, pat, id)),
    $x \neq y$

Once a user has activated the `Third-party()` role, he can grant existing third-party consent requests by activating the corresponding `Third-party-consent`(x, pat, id) role (S2.2.14):

(S2.2.14)
canActivate(x, `Third-party-consent`(x, pat, id)) ←
    hasActivated(x, `Third-party()`),
    hasActivated(y, `Request-third-party-consent`(x, pat, id))

Alternatively, consent can also be given by a clinician currently treating the patient to whom the record item belongs (S2.2.15), as in many cases the third party will not be able to give consent in this way, or the treating clinician can deduce the third party's implied consent:

(S2.2.15)
canActivate(cli, `Third-party-consent`(x, pat, id)) ←
    hasActivated(cli, `Spine-clinician`($ra, org, spcty$)),
    canActivate(cli, `Treating-clinician`($pat, org, spcty$)),
    hasActivated(y, `Request-third-party-consent`(x, pat, id))

Third-party consent is automatically cancelled if all relevant requests have been cancelled

(S2.2.16) (hence the consent role never needs to be deactivated directly):

(S2.2.16)
isDeactivated(x, Third-party-consent(x, pat, id)) ←
    isDeactivated(y, Request-third-party-consent(x, pat, id)),
    other-third-party-consent-requests(0, y, x)

Consent to Treatment The OBS does not clearly say how legitimate relationships are formally established. Our policy requires the patient to give express consent for a clinician to have the role of a *treating clinician* that represents the legitimate relationship between the clinician and the patient.

As is the case with third-party consent, the consent to treatment of patient *pat* by clinician *cli* must first be requested by a clinician (this may be *cli* herself), in order to prevent non-professional users from activating a large number of unrequested consent roles. The request is represented by the activation of Request-consent-to-treatment(*pat*, *org*, *cli*, *spcty*), where *org* is *cli*'s health organisation and *spcty* her specialty (S2.3.1):

(S2.3.1)
canActivate($cli1$, Request-consent-to-treatment(pat, $org2$, $cli2$, $spcty2$)) ←
    hasActivated($cli1$, Spine-clinician($ra1$, $org1$, $spcty1$)),
    canActivate($cli2$, Spine-clinician($ra2$, $org2$, $spcty2$)),
    canActivate(pat, Patient())

The request can be cancelled by the requester herself (S2.3.2), by the clinician *cli* (S2.3.3), by the patient (S2.3.4), or his agent (S2.3.5) or his GP (S2.3.6) on the patient's behalf:

(S2.3.2)
canDeactivate($cli1$, $cli1$,
        Request-consent-to-treatment(pat, $org2$, $cli2$, $spcty2$)) ←
    hasActivated($cli1$, Spine-clinician($ra1$, $org1$, $spcty1$))

(S2.3.3)
canDeactivate($cli2$, $cli1$,
        Request-consent-to-treatment(pat, $org2$, $cli2$, $spcty2$)) ←
    hasActivated($cli2$, Spine-clinician($ra2$, $org2$, $spcty2$))

(S2.3.4)
canDeactivate(pat, x, Request-consent-to-treatment(pat, org, cli, $spcty$)) ←
    hasActivated(pat, Patient())

(S2.3.5)
canDeactivate(ag, x, Request-consent-to-treatment(pat, org, cli, $spcty$)) ←
    hasActivated(ag, Agent(pat))

(S2.3.6)
canDeactivate(cli, x, Request-consent-to-treatment(pat, org, $cli2$, $spcty$)) ←
    hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
    canActivate(cli, General-practitioner(pat))

A request is automatically deactivated when the patient's registration is cancelled (S2.3.7):

(S2.3.7)
isDeactivated(x, `Request-consent-to-treatment`($pat, org, cli, spcty$)) ←
    isDeactivated(y, `Register-patient`(pat))

An activated request can be granted by the patient directly (S2.3.9), or by his agent (S2.3.10) or any treating clinician (S2.3.11) on his behalf, by activating `Consent-to-treatment`(*pat*, *org*, *cli*, *spcty*):

(S2.3.9)
canActivate(pat, `Consent-to-treatment`($pat, org, cli, spcty$)) ←
    hasActivated(pat, `Patient`()),
    hasActivated(x, `Request-consent-to-treatment`($pat, org, cli, spcty$))

(S2.3.10)
canActivate(ag, `Consent-to-treatment`($pat, org, cli, spcty$)) ←
    hasActivated(ag, `Agent`(pat)),
    hasActivated(x, `Request-consent-to-treatment`($pat, org, cli, spcty$))

(S2.3.11)
canActivate($cli1$, `Consent-to-treatment`($pat, org, cli2, spcty$)) ←
    hasActivated($cli1$, `Spine-clinician`($ra, org, spcty$)),
    canActivate($cli1$, `Treating-clinician`($pat, org, spcty$)),
    hasActivated(x, `Request-consent-to-treatment`($pat, org, cli2, spcty$))

Consent is automatically cancelled if all relevant requests have been withdrawn (S2.3.12, S2.3.8):

(S2.3.12)
isDeactivated(x, `Consent-to-treatment`($pat, org, cli, spcty$)) ←
    isDeactivated(y, `Request-consent-to-treatment`($pat, org, cli, spcty$)),
    other-consent-to-treatment-requests($0, y, pat, org, cli, spcty$)

(S2.3.8)
other-consent-to-treatment-requests(count$\langle y \rangle, x, pat, org, cli, spcty$) ←
    hasActivated(y, `Request-consent-to-treatment`($pat, org, cli, spcty$)),
    $x \neq y$

Often, it is a workgroup or team consisting of several clinicians providing care to the patient, each requiring access to the patient's record (§730.20.2). To support workgroup-based authorisation, patients can activate the `Consent-to-group-treatment`(*pat*, *org*, *group*) role if the corresponding `Request-consent-to-group-treatment` role has been activated. The relevant rules (S2.4.1–12) are much the same as those for standard consent to treatment except that the request can also be deactivated by workgroup members (S2.4.6). Workgroup membership is checked by requesting an RA-issued membership cre-

dential from the RA:

(S2.4.6)
canDeactivate(cli, x, `Request-consent-to-group-treatment`($pat, org, group$)) ←
    hasActivated(cli, `Spine-clinician`($ra, org, spcty$)),
    ra◇ra.canActivate(cli, `Workgroup-member`($org, group, spcty$))

Legitimate Relationship

In Phase 3, only clinicians with legitimate relationships will have access to a patient's data (§730.17). We have seen above how such a relationship is formed by the clinician requesting the patient's consent and the patient giving consent to treatment. Here we discuss some more ways for establishing legitimate relationships, and the rules concerning the auxiliary roles `Treating-clinician` and `Group-treating-clinician`.

Referrals Clinicians can "delegate" their legitimate relationship to a patient to another clinician by the act of referral (§730.20.8). No express patient consent is needed in this case.

A clinician currently treating a patient *pat* can refer the patient to another clinician *cli* from *org* in specialty *spcty* by activating `Referrer`(*pat*, *org*, *cli*, *spcty*) (S3.1.1):

(S3.1.1)
canActivate($cli1$, `Referrer`($pat, org, cli2, spcty1$)) ←
    hasActivated($cli1$, `Spine-clinician`($ra, org, spcty2$)),
    canActivate($cli1$, `Treating-clinician`($pat, org, spcty2$))

Both the referring clinician (S3.1.2) and the patient (S3.1.3) can cancel the referral:

(S3.1.2)
canDeactivate($cli1, cli1$, `Referrer`($pat, org, cli2, spcty1$)) ←

(S3.1.3)
canDeactivate($pat, cli1$, `Referrer`($pat, org, cli2, spcty1$)) ←

The referral role is also deactivated if the patient's registration is cancelled (S3.1.4):

(S3.1.4)
isDeactivated($cli1$, `Referrer`($pat, org, cli2, spcty1$)) ←
    isDeactivated(x, `Register-patient`(pat))

Accident and Emergency In the case of accident or emergency, it may be necessary to get access to a patient's records without his explicit consent. A clinician can activate the `Spine-emergency-clinician`(*org*, *pat*) role for any registered patient *pat* (S3.2.1):

(S3.2.1)
canActivate(cli, `Spine-emergency-clinician`(org, pat)) ←
    hasActivated(cli, `Spine-clinician`($ra, org, spcty$)),
    canActivate(pat, `Patient`())

This should trigger an alert and be highlighted in the audit trail (notifications and audit are not modelled by our policies). The `Spine-emergency-clinician` role can be deactivated

by the clinician herself (S3.2.2):

(S3.2.2)
canDeactivate(cli, cli, `Spine-emergency-clinician`(org, pat)) ←

The role is automatically deactivated if the user's clinician role is deactivated (S3.2.3) and also if the patient's registration is cancelled (S3.2.4):

(S3.2.3)
isDeactivated(x, `Spine-emergency-clinician`(org, pat)) ←
    isDeactivated(x, `Spine-clinician`($ra, org, spcty$))

(S3.2.4)
isDeactivated(x, `Spine-emergency-clinician`(org, pat)) ←
    isDeactivated(y, `Register-patient`(pat))

Treating Clinicians A legitimate relationship between a clinician or a workgroup and a patient is manifested in the roles `Treating-clinician` or `Group-treating-clinician`, respectively. These are auxiliary roles that never need to be actually activated as it is only ever checked whether a user *can* activate them, but not whether they *have* been activated.

A clinician *cli* is associated with the role `Treating-clinician`(*pat*,*org*,*spcty*) if express consent to treatment has been given, i.e. the matching role `Consent-to-treatment`(*pat*, *org*, *cli*, *spcty*) has been activated (S3.3.1):

(S3.3.1)
canActivate(cli, `Treating-clinician`($pat, org, spcty$)) ←
    hasActivated(x, `Consent-to-treatment`($pat, org, cli, spcty$))

Alternatively, no consent is required if the clinician is active as `Spine-emergency-clinician`(*org*, *pat*), but in this case, *spcty* must be set to "`A-and-E`" (S3.3.2):

(S3.3.2)
canActivate(cli, `Treating-clinician`($pat, org, spcty$)) ←
    hasActivated(cli, `Spine-emergency-clinician`(org, pat)),
    $spcty =$ `A-and-E`

Treating clinicians are also those with a matching referral (S3.3.3):

(S3.3.3)
canActivate(cli, `Treating-clinician`($pat, org, spcty$)) ←
    canActivate(cli, `Spine-clinician`($ra, org, spcty$)),
    hasActivated(x, `Referrer`($pat, org, cli, spcty$))

The GP role `General-practitioner`(pat) is derived from the `Treating-clinician` role and represents a clinician who treats that patient in the specialty "`GP`" (S3.3.5):

(S3.3.5)
canActivate(cli, `General-practitioner`(pat)) ←
    canActivate(cli, `Treating-clinician`($pat, org, spcty$)),
    $spcty =$ `GP`

Similarly, the `Group-treating-clinician` role can be activated by members of the workgroup, if consent to group treatment has been given. An RA-issued credential is required for proving workgroup membership, either to be submitted directly (S3.4.1) or automatically fetched from the clinician's RA (S3.4.2). The rules also check whether the RA is approved by the NHS:

(S3.4.1)
canActivate(*cli*, `Group-treating-clinician`(*pat*, *ra*, *org*, *group*, *spcty*)) ←
    hasActivated(*x*, `Consent-to-group-treatment`(*pat*, *org*, *group*)),
    ra.canActivate(*cli*, `Workgroup-member`(*org*, *group*, *spcty*)),
    canActivate(*ra*, `Registration-authority`())

(S3.4.2)
canActivate(*cli*, `Group-treating-clinician`(*pat*, *ra*, *org*, *group*, *spcty*)) ←
    hasActivated(*x*, `Consent-to-group-treatment`(*pat*, *org*, *group*)),
    ra◇*ra*.canActivate(*cli*, `Workgroup-member`(*org*, *group*, *spcty*)),
    canActivate(*ra*, `Registration-authority`())

Any `Group-treating-clinician` can also be a `Treating-clinician` (S3.3.4); this is a simple example of role hierarchy:

(S3.3.4)
canActivate(*cli*, `Treating-clinician`(*pat*, *org*, *spcty*)) ←
    canActivate(*cli*, `Group-treating-clinician`(*pat*, *ra*, *org*, *group*, *spcty*))

Sealing Off Data

In the past, patients were often refused access to their own records. The recent decades have brought much more openness between doctors and patients [SMW93]. The change is also reflected in law: the Medical Reports Act 1988 and the Access to Health Records Act 1990 give patients the right to access records created after November 1991. There still are, however, exceptional situations in which doctors may withhold parts of the record. The legislation states two legitimate reasons: firstly, if the information relates to third parties, and secondly, when the information must be regarded as harmful to the patient.

Clinicians Restricting Access The Spine will have provisions for clinicians to seal off data from the patient in exceptional circumstances (§730.49). We have already showed how our policy deals with record items relating to third parties. Treating clinicians can seal off a set of items *ids* from a patient *pat*'s EHR by activating the role `Concealed-by-spine-clinician`(*pat*, *ids*, *start*, *end*) (S4.1.1):

(S4.1.1)
canActivate(*cli*, `Concealed-by-spine-clinician`(*pat*, *ids*, *start*, *end*)) ←
    hasActivated(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)),
    canActivate(*cli*, `Treating-clinician`(*pat*, *org*, *spcty*))

The access restriction is valid only within the time interval [*start*, *end*] (cf. §730.51.8). Any such access restriction can be lifted by the clinician who imposed it (S4.1.2), by the patient's

GP (S4.1.3), or by any clinician working in the same team (S4.1.4) (cf. §730.51.10):

(S4.1.2)
canDeactivate($cli, cli,$ `Concealed-by-spine-clinician`$(pat, ids, start, end)) \leftarrow$
hasActivated($cli,$ `Spine-clinician`$(ra, org, spcty)$)

(S4.1.3)
canDeactivate($cli, cli2,$ `Concealed-by-spine-clinician`$(pat, ids, start, end)) \leftarrow$
hasActivated($cli,$ `Spine-clinician`$(ra, org, spcty)$),
canActivate($cli,$ `General-practitioner`(pat))

(S4.1.4)
canDeactivate($cli1, cli2,$ `Concealed-by-spine-clinician`$(pat, ids, start, end)) \leftarrow$
hasActivated($cli1,$ `Spine-clinician`$(ra, org, spcty1)$),
canActivate($cli1,$ `Group-treating-clinician`$(pat, ra, org, group, spcty1)$),
canActivate($cli2,$ `Group-treating-clinician`$(pat, ra, org, group, spcty2)$),
hasActivated($x,$ `Consent-to-group-treatment`$(pat, org, group)$)

The role is automatically deactivated as soon as the patient's registration is cancelled (S4.1.5):

(S4.1.5)
isDeactivated($x,$ `Concealed-by-spine-clinician`$(pat, ids, start, end)) \leftarrow$
isDeactivated($y,$ `Register-patient`(pat))

Patients Restricting Access The OBS also allows patients to seal off selected clinical parts of their record (§730.45). The specification suggests that the patient should file a sealing-off request which is subsequently dealt with by a clinician (§730.48.2).

Our solution gives patients much flexibility for specifying record items to seal off. A patient (S4.2.1) or his agent (S4.2.2) can file a sealing-off request by activating the role `Conceal-request`(*which, who, start, end*):

(S4.2.1)
canActivate($pat,$ `Conceal-request`$(what, who, start, end)) \leftarrow$
hasActivated($pat,$ `Patient`$()$),
count-conceal-requests(n, pat),
$what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time)$,
$who = (orgs1, readers1, spctys1)$,
$n < 100$

(S4.2.2)
canActivate($ag,$ `Conceal-request`$(what, who, start, end)) \leftarrow$
hasActivated($ag,$ `Agent`(pat)),
count-conceal-requests(n, pat),
$what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time)$,
$who = (orgs1, readers1, spctys1)$,
$n < 100$

For each patient, a maximum of 100 such requests can be activated (S4.2.7):

(S4.2.7)
count-conceal-requests(count⟨y⟩, pat) ←
    hasActivated(x, `Conceal-request`(y)),
    $what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time)$,
    $who = (orgs1, readers1, spctys1)$,
    $y = (what, who, start, end)$

Again, we invented this common-sense rule to prevent non-professional users from clogging up the policy.

Apart from a validity time interval [*start*, *end*] (cf. 730.48.12), the `Conceal-request` role specifies which items to seal off and whom the restriction applies to. The specification for *which* items to seal off is a 7-tuple

$$(pat, ids, orgs, authors, subjects, from\text{-}time,\ to\text{-}time).$$

A record item from a patient *pat*'s EHR is sealed off if its ID is in the set *ids*, its author is in the set *authors* and working for a health organisation in *orgs*, if its subject matter is in *subjects*, and its creation date is between *from-time* and *to-time*.

The specification for *whom* the restriction applies to is a triple

$$(orgs, readers, spctys).$$

A user is prevented from accessing the selected items (even if she has a legitimate relationship) if she is in the set *readers*, working for a health organisation (if applicable) in *orgs* in a specialty (if applicable) in the set *spctys*.

With $\mathcal{C}_0$'s universal set expression Ω and the set difference construct, patients can express explicit access permissions (e.g. *"only doctors from Addenbrooke's may access items concerning cancer"*) as well as explicit access denials (e.g. *"Dr Littlewood may not access items created after 2005"*). Access to items that have not yet been created can also be restricted by setting *ids* to the universal set expression for IDs, and setting *to-time* to the future.

The `Conceal-request` role can also be used to request to withhold items from non-clinicians, including agents and the patient himself[8].

The patient (S4.2.3), his agents (S4.2.4) and his GP (S4.2.5) can all deactivate requests:

(S4.2.3)
canDeactivate(pat, x, `Conceal-request`($what, whom, start, end$)) ←
    hasActivated(pat, `Patient`()),
    $\pi_1^7(what) = pat$

(S4.2.4)
canDeactivate(ag, x, `Conceal-request`($what, whom, start, end$)) ←
    hasActivated(ag, `Agent`(pat)),
    $\pi_1^7(what) = pat$

[8] Some patients wish not to be informed about certain particularly distressing subject matters. For example, a patient may specify to make all record items regarding cancer, including those created in the future, inaccessible to himself.

(S4.2.5)
canDeactivate(cli, x, Conceal-request($what, whom, start, end$)) ←
hasActivated(cli, Spine-clinician($ra, org, spcty$)),
canActivate(cli, General-practitioner(pat)),
$\pi_1^7(what) = pat$

A request is automatically deactivated if the patient's registration is cancelled (S4.2.6):

(S4.2.6)
isDeactivated(x, Conceal-request($what, whom, start, end$)) ←
isDeactivated(y, Register-patient(pat)),
$\pi_1^7(what) = pat$

A treating clinician can apply the request by activating a matching Concealed-by-spine-patient role (S4.2.8):

(S4.2.8)
canActivate(cli, Concealed-by-spine-patient($what, who, start, end$)) ←
hasActivated(cli, Spine-clinician($ra, org, spcty$)),
canActivate(cli, Treating-clinician($pat, org, spcty$)),
hasActivated(x, Conceal-request($what, who, start, end$))

This role can be revoked by the activator herself (S4.2.9) or any other clinician working in the same workgroup (S4.2.10):

(S4.2.9)
canDeactivate(cli, cli, Concealed-by-spine-patient($what, who, start, end$)) ←
hasActivated(cli, Spine-clinician($ra, org, spcty$))

(S4.2.10)
canDeactivate($cli1, cli2$, Concealed-by-spine-patient($what, who, start1, end1$)) ←
hasActivated($cli1$, Spine-clinician($ra, org, spcty1$)),
$ra \Diamond ra$.canActivate($cli1$,
Group-treating-clinician($pat, ra, org, group, spcty1$)),
$ra \Diamond ra$.canActivate($cli2$,
Group-treating-clinician($pat, ra, org, group, spcty2$))

The role is automatically revoked if the request is cancelled (S4.2.11):

(S4.2.11)
isDeactivated(cli, Concealed-by-spine-patient($what, who, start, end$)) ←
isDeactivated(x, Conceal-request($what, who, start, end$))

This last rule in combination with S4.2.5 means that the patient's GPs can always remove an access restriction.

A clinician who is granted *authenticated express consent* by a patient may access any sealed-off data without raising an alert if she would also be entitled to access that data, had it not bean sealed off (§730.48.4, §730.48.17). A patient *pat* (S4.3.1), his agent (S4.3.2), or his GP (S4.3.3) can activate the role Authenticated-express-consent(*pat, cli*) for a

clinician *cli*:

(S4.3.1)
canActivate(pat, Authenticated-express-consent(pat, cli)) ←
    hasActivated(pat, Patient()),
    count-authenticated-express-consent(n, pat),
    $n < 100$

(S4.3.2)
canActivate(ag, Authenticated-express-consent(pat, cli)) ←
    hasActivated(ag, Agent(pat)),
    count-authenticated-express-consent(n, pat),
    $n < 100$

(S4.3.3)
canActivate($cli1$, Authenticated-express-consent(pat, $cli2$)) ←
    hasActivated($cli1$, Spine-clinician(ra, org, $spcty$)),
    canActivate($cli1$, General-practitioner(pat))

The following aggregation rule restricts the number of such consent grants to 100 for the former two cases of activations (S4.3.8):

(S4.3.8)
count-authenticated-express-consent(count⟨cli⟩, pat) ←
    hasActivated(x, Authenticated-express-consent(pat, cli))

Similarly, consent can be withdrawn by the patient (S4.3.4), his agent(S4.3.5), or his GP (S4.3.6):

(S4.3.4)
canDeactivate(pat, x, Authenticated-express-consent(pat, cli)) ←
    hasActivated(pat, Patient())

(S4.3.5)
canDeactivate(ag, x, Authenticated-express-consent(pat, cli)) ←
    hasActivated(ag, Agent(pat))

(S4.3.6)
canDeactivate($cli1$, x, Authenticated-express-consent(pat, $cli2$)) ←
    hasActivated($cli1$, Spine-clinician(ra, org, $spcty$)),
    canActivate($cli1$, General-practitioner(pat))

It is automatically withdrawn when the patient's registration is cancelled (S4.3.7):

(S4.3.7)
isDeactivated(x, Authenticated-express-consent(pat, cli)) ←
    isDeactivated(y, Register-patient(pat))

Access Permissions

A new record item for a patient can be created by treating clinicians performing the action `Add-spine-record-item`(*pat*) (S5.1.1):

(S5.1.1)
permits(*cli*, `Add-spine-record-item`(*pat*)) ←
    hasActivated(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)),
    canActivate(*cli*, `Treating-clinician`(*pat*, *org*, *spcty*))

Patients may not add items to their records themselves, but they can add comments to existing items by performing `Annotate-spine-record-item`(*pat, id*) (S5.1.2):

(S5.1.2)
permits(*pat*, `Annotate-spine-record-item`(*pat*, *id*)) ←
    hasActivated(*pat*, `Patient`())

Comments can also be added by a patient's agent (S5.1.3) or a treating clinician (S5.1.4) on his behalf (§730.59.6):

(S5.1.3)
permits(*ag*, `Annotate-spine-record-item`(*pat*, *id*)) ←
    hasActivated(*ag*, `Agent`(*pat*))

(S5.1.4)
permits(*pat*, `Annotate-spine-record-item`(*pat*, *id*)) ←
    hasActivated(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)),
    canActivate(*cli*, `Treating-clinician`(*pat*, *org*, *spcty*))

Patients (S5.2.1), their agents (S5.2.2) and treating clinicians (S5.2.3) can get a list of all record item IDs of the patient by performing the action `Get-spine-record-item-ids`(*pat*):

(S5.2.1)
permits(*pat*, `Get-spine-record-item-ids`(*pat*)) ←
    hasActivated(*pat*, `Patient`())

(S5.2.2)
permits(*ag*, `Get-spine-record-item-ids`(*pat*)) ←
    hasActivated(*ag*, `Agent`(*pat*))

(S5.2.3)
permits(*cli*, `Get-spine-record-item-ids`(*pat*)) ←
    hasActivated(*cli*, `Spine-clinician`(*ra*, *org*, *spcty*)),
    canActivate(*cli*, `Treating-clinician`(*pat*, *org*, *spcty*))

Depending on system implementation, each ID may be annotated with additional, non-confidential information about that item, such as date etc.

A patient (S5.3.1) or his agent (S5.3.2) can read a record item from the patient's record by performing the action `Read-spine-record-item`(*pat, id*) if the patient has given one-off

consent to make his data available:

(S5.3.1)
permits(pat, Read-spine-record-item(pat, id)) ←
hasActivated(pat, Patient()),
hasActivated(x, One-off-consent(pat)),
count-concealed-by-spine-patient(n, a, b),
count-concealed-by-spine-clinician(m, pat, id),
third-party-consent($consenters$, pat, id),
$n = 0$,
$m = 0$,
$a = (pat, id)$,
$b =$ (No-org, pat, No-spcty),
Get-spine-record-third-parties(pat, id) $\subseteq$ $consenters$

(S5.3.2)
permits(ag, Read-spine-record-item(pat, id)) ←
hasActivated(ag, Agent(pat)),
hasActivated(x, One-off-consent(pat)),
count-concealed-by-spine-patient(n, a, b),
count-concealed-by-spine-clinician(m, pat, id),
third-party-consent($consenters$, pat, id),
$n = 0$,
$m = 0$,
$a = (pat, id)$,
$b =$ (No-org, ag, No-spcty),
Get-spine-record-third-parties(pat, id) $\subseteq$ $consenters$

The former rules authorise read access only if the items have not been sealed off. The check is implemented by the use of aggregation rules (S4.1.6, S4.2.12):

(S4.1.6)
count-concealed-by-spine-clinician(count$\langle x \rangle$, pat, id) ←
hasActivated(x, Concealed-by-spine-clinician(pat, ids, $start$, end)),
$id \in ids$,
Current-time() $\in [start, end]$

(S4.2.12)
count-concealed-by-spine-patient(count$\langle x\rangle, a, b$) $\leftarrow$
hasActivated(x, `Concealed-by-spine-patient`($what, who, start, end$)),
$a = (pat, id)$,
$b = (org, reader, spcty)$,
$what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time)$,
$whom = (orgs1, readers1, spctys1)$,
`Get-spine-record-org`(pat, id) $\in orgs$,
`Get-spine-record-author`(pat, id) $\in authors$,
$sub \in$ `Get-spine-record-subjects`(pat, id),
$sub \in subjects$,
`Get-spine-record-time`(pat, id) $\in [from\text{-}time, to\text{-}time]$,
$id \in ids$,
$org \in orgs1$,
$reader \in readers1$,
$spcty \in spctys1$,
`Current-time`() $\in [start, end]$,
`Get-spine-record-third-parties`(pat, id) $= \{\}$,
`non-clinical` $\in \Omega -$ `Get-spine-record-subjects`(pat, id)

It is also checked that all relevant third parties have given consent to disclosure (S2.2.17)[9] (§730.20.9, §730.56):

(S2.2.17)
third-party-consent(group$\langle consenter\rangle, pat, id$) $\leftarrow$
hasActivated(x, `Third-party-consent`($consenter, pat, id$))

The author of a record item can always read it herself as long the patient has given his one-off consent, even if it has been sealed off by the patient (S5.3.3):

(S5.3.3)
permits(cli, `Read-spine-record-item`(pat, id)) $\leftarrow$
hasActivated(cli, `Spine-clinician`($ra, org, spcty$)),
hasActivated(x, `One-off-consent`(pat)),
`Get-spine-record-org`(pat, id) $= org$,
`Get-spine-record-author`(pat, id) $= cli$

A treating clinician may view the item if the patient has given his one-off consent, if it has not been sealed off by the patient and if her specialty allows her[10] to read items regarding

[9] Note that third-party-consent is a predicate name, whereas `Third-party-consent` is a role name.

[10] We associate a set of permitted subjects for each specialty with the built-in function `Permitted-subjects`(*spcty*). For example, the specialty "dentistry" may allow the subjects "general" and "dental", but not "venereal disease".

the subject-matters of the item (S5.3.4):

(S5.3.4)
permits(cli, Read-spine-record-item(pat, id)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
hasActivated(x, One-off-consent(pat)),
canActivate(cli, Treating-clinician(pat, org, $spcty$)),
count-concealed-by-spine-patient(n, a, b),
$n = 0$,
$a = (pat, id)$,
$b = (org, cli, spcty)$,
Get-spine-record-subjects(pat, id) $\subseteq$ Permitted-subjects($spcty$)

If the item is sealed off by the patient, she is only permitted to read it with authenticated express consent (S5.3.5) (§730.48.21).

(S5.3.5)
permits(cli, Read-spine-record-item(pat, id)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
hasActivated(x, One-off-consent(pat)),
canActivate(cli, Treating-clinician(pat, org, $spcty$)),
hasActivated(y, Authenticated-express-consent(pat, cli)),
Get-spine-record-subjects(pat, id) $\subseteq$ Permitted-subjects($spcty$)

The OBS (§730.48.17, §730.4.11) allows clinicians with a legitimate relationship to access a patient's item even if it has been sealed off by the patient, and even if the patient has not given any one-off consent (S5.3.6):

(S5.3.6)
permits(cli, Force-read-spine-record-item(pat, id)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
canActivate(cli, Treating-clinician(pat, org, $spcty$))

Of course, clinicians are meant to "break the seal" and access the item only in exceptional circumstances. The action Force-read-spine-record-item(*pat, id*) should trigger an alert in a real implementation and be marked in the audit trail. There is no override facility for patients to access data that has been sealed off by a clinician (§730.51.14).

9.4.3 Patient Demographic Service

The PDS will provide users with various search functions on patient demographic data. Our PDS policy only implements the minimal functionality required to interoperate with the Spine.

Main Roles and Patient Registration

The main roles on the PDS are PDS-manager, Patient, Agent, and Professional-user. As before, a user may be active in only one such role at a time. This separation-of-duties con-

straint is enforced by the use of aggregation rules (P1.5.1, P1.1.4, P1.2.4, P1.3.5, P1.4.6):

(P1.5.1)
no-main-role-active($user$) ←
count-agent-activations(n,$user$),
count-patient-activations(n,$user$),
count-PDS-manager-activations(n,$user$),
count-preofessional-user-activations(n,$user$),
$n = 0$

(P1.1.4)
count-PDS-manager-activations(count⟨u⟩,$user$) ←
hasActivated($user$,PDS-manager())

(P1.2.4)
count-patient-activations(count⟨u⟩,$user$) ←
hasActivated($user$,Patient())

(P1.3.5)
count-agent-activations(count⟨u⟩,$user$) ←
hasActivated($user$,Agent(pat))

(P1.4.6)
count-professional-user-activations(count⟨u⟩,$user$) ←
hasActivated($user$,Professional-user(ra,org))

Any user active in a role can deactivate their own role (P1.1.2, P1.2.2, P1.3.2, P1.4.5):

(P1.1.2)
canDeactivate(adm,adm,PDS-manager()) ←

(P1.2.2)
canDeactivate(pat,pat,Patient()) ←

(P1.3.2)
canDeactivate(ag,ag,Agent(pat)) ←

(P1.4.5)
canDeactivate(x,x,Professional-user(ra,org)) ←

A user can activate the PDS-manager() role if she is registered as a manager (P1.1.1):

(P1.1.1)
canActivate(adm,PDS-manager()) ←
hasActivated(x,Register-PDS-manager(adm)),
no-main-role-active(adm)

A manager can delegate the PDS-manager role to another user *usr* by activating Register-PDS-manager(*usr*) if *usr* has not already been registered by another manager (P1.1.5,

P1.1.7):

(P1.1.5)
canActivate($adm1$, Register-PDS-manager($adm2$)) ←
    hasActivated($adm1$, PDS-manager()),
    pds-admin-regs($0, adm2$)

(P1.1.7)
pds-admin-regs(count$\langle x \rangle, adm$) ←
    hasActivated(x, Register-PDS-manager(adm))

Managers can also cancel manager registrations (P1.1.6), thereby potentially triggering the deactivation of any matching active manager role (P1.1.3):

(P1.1.6)
canDeactivate($adm1, x$, Register-PDS-manager($adm2$)) ←
    hasActivated($adm1$, PDS-manager())

(P1.1.3)
isDeactivated(adm, PDS-manager()) ←
    isDeactivated(x, Register-PDS-manager(adm))

In our policy, the manager role is designated for registering patients and storing their demographic data. A manager can register a patient *pat* who has not yet been registered so far by activating Register-patient(*pat*) (P2.1.1, P2.1.3):

(P2.1.1)
canActivate(adm, Register-patient(pat)) ←
    hasActivated(adm, PDS-manager()),
    patient-regs($0, pat$)

(P2.1.3)
patient-regs(count$\langle x \rangle, pat$) ←
    hasActivated(x, Register-patient(pat))

Every patient in the country will be associated with such a registration role activation. Managers can also deactivate patient registrations (P2.1.2):

(P2.1.2)
canDeactivate(adm, x, Register-patient(pat)) ←
    hasActivated(adm, PDS-manager())

Patients can activate the Patient() role on the PDS if they are registered. An agent can activate the Agent(*pat*) role if he himself is a registered patient, and the Spine confirms that he is an agent (P1.3.1):

(P1.3.1)
canActivate(ag, Agent(pat)) ←
    hasActivated(x, Register-patient(ag)),
    no-main-role-active(ag),
    Spine◊Spine.canActivate(ag, Agent(pat))

Patient and their agent roles are deactivated if their registrations are cancelled (P1.2.3, P1.3.3, P1.3.4):

(P1.2.3)
isDeactivated(pat, `Patient`()) ←
    isDeactivated(x, `Register-patient`(pat))

(P1.3.3)
isDeactivated(ag, `Agent`(pat)) ←
    isDeactivated(x, `Register-patient`(ag))

(P1.3.4)
isDeactivated(ag, `Agent`(pat)) ←
    isDeactivated(x, `Register-patient`(pat))

Our policy only exemplifies two kinds of professional users, clinicians and Caldicott Guardians, but other professional roles could easily be implemented in a similar fashion. A user submitting a currently valid RA-issued clinician or Caldicott Guardian credential can activate the `Professional-user`(*ra, org*) role (P1.4.1, P1.4.3):

(P1.4.1)
canActivate(x, `Professional-user`(ra, org)) ←
    no-main-role-active(cli),
    ra.hasActivated(x, `NHS-clinician-cert`($org, cli, spcty, start, end$)),
    canActivate(ra, `Registration-authority`()),
    `Current-time`() ∈ [$start, end$]

(P1.4.3)
canActivate(x, `Professional-user`(ra, org)) ←
    no-main-role-active(cg),
    ra.hasActivated(x, `NHS-Caldicott-guardian-cert`($org, cg, start, end$)),
    canActivate(ra, `Registration-authority`()),
    `Current-time`() ∈ [$start, end$]

If no valid credential is submitted, it is requested directly from the RA (P1.4.2, P1.4.4).

(P1.4.2)
canActivate(x, `Professional-user`(ra, org)) ←
    no-main-role-active(cli),
    ra◇ra.hasActivated(x, `NHS-clinician-cert`($org, cli, spcty, start, end$)),
    canActivate(ra, `Registration-authority`()),
    `Current-time`() ∈ [$start, end$]

(P1.4.4)
canActivate(x, `Professional-user`(ra, org)) ←
    no-main-role-active(cg),
    ra◇ra.hasActivated(x, `NHS-Caldicott-guardian-cert`($org, cg, start, end$)),
    canActivate(ra, `Registration-authority`()),
    `Current-time`() ∈ [$start, end$]

In all cases, it is checked whether the RA is approved by the NHS (P1.5.2, P1.5.3):

(P1.5.2)
canActivate(ra, Registration-authority()) ←
NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end)),
Current-time() ∈ [$start$, end]

(P1.5.3)
canActivate(ra, Registration-authority()) ←
ra◇NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end)),
Current-time() ∈ [$start$, end]

Patient-Registration Credentials

Patients are authenticated on the Spine and other applications after the PDS is contacted for confirmation of the patient's registration status. A request sent to the PDS for this purpose asks for the credential of the form PDS.hasActivated(x, Register-patient(pat)).

The PDS policy allows patient-registration-credential requests from patients (P2.2.1), agents (P2.2.2) and professional users (P2.2.3) who have activated their respective roles:

(P2.2.1)
canReqCred(pat, PDS.hasActivated(x, Register-patient(pat))) ←
hasActivated(pat, Patient())

(P2.2.2)
canReqCred(ag, PDS.hasActivated(x, Register-patient(pat))) ←
hasActivated(ag, Agent(pat))

(P2.2.3)
canReqCred(usr, PDS.hasActivated(x, Register-patient(pat))) ←
hasActivated(usr, Professional-user(ra, org))

Credential requests are further granted to health organisations certified by an RA (P2.2.4). If no RA-issued health organisation credential is submitted, the PDS will try to retrieve it from the health organisation directly (P2.5.5):

(P2.2.4)
canReqCred(org, PDS.hasActivated(x, Register-patient(pat))) ←
ra.hasActivated(x, NHS-health-org-cert(org, $start$, end)),
canActivate(ra, Registration-authority())

(P2.2.5)
canReqCred(org, PDS.hasActivated(x, Register-patient(pat))) ←
org◇ra.hasActivated(x, NHS-health-org-cert(org, $start$, end)),
canActivate(ra, Registration-authority())

Lastly, patient-registration credentials can also be revealed to RAs (P2.2.6) and the Spine

(P2.2.7):

(P2.2.6)
canReqCred(ra, PDS.hasActivated(x, Register-patient(pat))) ←
    canActivate(ra, Registration-authority())

(P2.2.7)
canReqCred(Spine, PDS.hasActivated(x, Register-patient(pat))) ←

9.4.4 Local Health Organisations

The policy of our exemplary health organisation, Addenbrooke's Hospital (ADB), illustrates the authorisation principles of an EPR system. It also shows how a local application can collaborate with the Spine and other national services. For example, clinical workgroups can be managed locally, and local workgroup membership can be used to gain access to EHR items on the Spine. Or conversely, to be authenticated as a patient's agent on the hospital's system, the local policy can make use of the Spine's agent registration facilities.

As large parts of the ABD's policy are very similar to the Spine's, the following description will go into less detail and focus on the main differences. The full set of rules can be found in Appendix A.3.

Main Roles

ADB's policy defines seven main roles, Clinician, Caldicott-guardian, HR-manager, Receptionist, Patient, Agent, Ext-treating-clinician and Third-party. As in the policies for the other services, aggregation rules ensure that only one main role can be activated at a time (A1.7.1, A1.1.7, A1.2.7, A1.3.7, A1.4.7, A1.5.7, A1.6.4, A2.2.5, A2.3.11).

The staff roles

- Clinician(*spcty*) (A1.1.4–7),
- Caldicott-guardian() (A1.2.4–7),
- HR-manager() (A1.3.4–7), and
- Receptionist() (A1.4.4–7)

can be activated by a user if they have been registered (or appointed) by a human-resource manager. The corresponding registration roles are

- Register-clinician(*usr, spcty*) (A1.1.1–3),
- Register-Caldicott-guardian(*usr*) (A1.2.1–3),
- Register-HR-manager(*usr*) (A1.3.1–3), and
- Register-receptionist(*usr*) (A1.4.1–3).

Users in these staff roles can deactivate their own roles. Their roles are automatically deactivated when their corresponding registration role is deactivated, which can only be done by a human-resource manager.

Similarly, patients are registered by receptionists via the Register-patient(*pat*) role (A1.5.1–3) upon which they can activate their Patient() role (A1.5.4–7). The activation rule also checks if the patient is registered on the PDS.

Agents can be registered via the `Register-agent`(*usr*, *pat*) by both patients and Caldicott Guardians (A1.6.5–10), upon which the `Agent`(*pat*) role can be activated (A1.6.1–4). A user can also become an agent at ADB without registration if he is registered as an agent on the Spine (A1.6.2).

Caldicott Guardians

Caldicott Guardians are responsible for safeguarding the confidentiality of patient information in a health organisation. They are expected to check the audit trails for possible misconduct and to investigate events that trigger an alarm, e.g. a clinician reading a restricted item or assuming the role of an emergency clinician. They can also give consent on behalf of a patient, or, in exceptional circumstances, make decisions against the wishes of the patient.

In ADB's policy, a Caldicott Guardian has the power to lift access restrictions imposed by patients or clinicians (A4.1.4, A4.2.5):

(A4.1.4)
canDeactivate(cg, cli, `Concealed-by-clinician`(pat, id, $start$, end)) ←
    hasActivated(cg, `Caldicott-guardian`())

(A4.2.5)
canDeactivate(cg, x, `Concealed-by-patient`($what$, $whom$, $start$, end)) ←
    hasActivated(cg, `Caldicott-guardian`())

Caldicott Guardians can give consent to referral of patients (A2.1.10):

(A2.1.10)
canActivate(cg, `Consent-to-referral`(pat, ra, org, cli, $spcty$)) ←
    hasActivated(cg, `Caldicott-guardian`()),
    hasActivated(x, `Request-consent-to-referral`(pat, ra, org, cli, $spcty$))

They can further give consent to access third-party information (A2.3.4, A2.3.17):

(A2.3.4)
canActivate(cg, `Request-third-party-consent`(x, pat, id)) ←
    hasActivated(cg, `Caldicott-guardian`()),
    $x \in$ `Get-record-third-parties`(pat, id)

(A2.3.17)
canActivate(cg, `Third-party-consent`(x, pat, id)) ←
    hasActivated(cg, `Caldicott-guardian`()),
    hasActivated(y, `Request-third-party-consent`(x, pat, id))

They have the right to appoint and revoke agents for patients (A1.6.6, A1.6.8):

(A1.6.6)
canActivate(cg, `Register-agent`($agent$, pat)) ←
    hasActivated(cg, `Caldicott-guardian`()),
    canActivate(pat, `Patient`())

(A1.6.8)
canDeactivate(cg, x, Register-agent($agent, pat$)) ←
    hasActivated(cg, Caldicott-guardian())

Finally, Caldicott Guardians can revoke an emergency clinician's role (A3.7.3):

(A3.7.3)
canDeactivate(cg, cli, Emergency-clinician(pat)) ←
    hasActivated(cg, Caldicott-guardian())

Referrals and External Clinicians

When a patient is referred to an external clinician by a treating clinician in the health organisation (e.g. a GP referring a patient to see a specialist), that clinician may need access to the local EPR. In our policy for ADB, such a referral automatically establishes a legitimate relationship that enables the external clinician to access relevant and unrestricted items of the referred patient's EPR by activating the Ext-treating-clinician role (A5.3.5):

(A5.3.5)
permits(cli, Read-record-item(pat, id)) ←
    hasActivated(cli, Ext-treating-clinician($pat, ra, org, spcty$)),
    count-concealed-by-patient2(n, a, b),
    $n = 0$,
    $a = (pat, id)$,
    $b = (org, cli,$ Ext-group$, spcty)$,
    Get-record-subjects(pat, id) $\subseteq$ Permitted-subjects($spcty$)

In contrast to the referral mechanism on the Spine, ADB's policy requires explicit patient consent. The rationale behind this decision is that the EPR stored in the local health organisation is generally more detailed and possibly more sensitive than the shared EHR on the Spine. Any local clinician currently treating the patient can file a request Request-consent-to-referral(*pat, ra, org, cli, spcty*) to have the patient referred to an external clinician *cli* working for *org* in specialty *spcty* (A2.1.1):

(A2.1.1)
canActivate($cli1$, Request-consent-to-referral($pat, ra, org, cli2, spcty2$)) ←
    hasActivated($cli1$, Clinician($spcty1$)),
    canActivate($cli1$, ADB-treating-clinician($pat, team, spcty1$))

The request can be withdrawn by the clinician herself (A2.1.2), or denied by the patient (2.1.3), his agent (A2.1.4), or a Caldicott Guardian (A2.1.5):

(A2.1.2)
canDeactivate(cli, cli, Request-consent-to-referral($pat, ra, org, cli, spcty$)) ←
    hasActivated(cli, Clinician($spcty$))

(A2.1.3)
canDeactivate(pat, x, Request-consent-to-referral($pat, ra, org, cli, spcty$)) ←
    hasActivated(pat, Patient())

(A2.1.4)
canDeactivate(ag, x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(ag, `Agent`(pat))

(A2.1.5)
canDeactivate(cg, x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(cg, `Caldicott-guardian`())

All referral requests are automatically cancelled if the patient is unregistered (A2.1.6):

(A2.1.6)
isDeactivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    isDeactivated(y, `Register-patient`(pat))

A referral request can be granted by the patient or his agent activating a matching `Consent-to-referral` role (A2.1.8–9):

(A2.1.8)
canActivate(pat, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(pat, `Patient`()),
    hasActivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$))

(A2.1.9)
canActivate(pat, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(pat, `Agent`(pat)),
    hasActivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$))

Caldicott Guardians have the power to give consent on behalf of a patient, even against his wishes (A2.1.10):

(A2.1.10)
canActivate(cg, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(cg, `Caldicott-guardian`()),
    hasActivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$))

The consent role is automatically deactivated when all matching requests have been denied (A2.1.7, A2.1.11):

(A2.1.11)
isDeactivated(x, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    isDeactivated(y, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)),
    other-consent-to-referral-requests($0, y, pat, ra, org, cli, spcty$)

(A2.1.7)
other-consent-to-referral-requests(count$\langle y \rangle, x, pat, ra, org, cli, spcty$) ←
    hasActivated(y, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)),
    $x \neq y$

Once consent has been given, the external clinician can activate and deactivate her role (A2.2.2, A2.2.3):

(A2.2.2)
canActivate(cli, `Ext-treating-clinician`($pat, ra, org, spcty$)) ←
hasActivated(ref, `Consent-to-referral`($pat, ra, org, cli, spcty$)),
no-main-role-active(cli),
$ra \Diamond ra$.hasActivated(y, `NHS-clinician-cert`($org, cli, spcty, start, end$)),
canActivate(ra, `Registration-authority`())

(A2.2.3)
canDeactivate(cli, cli, `Ext-treating-clinician`($pat, ra, org, spcty$)) ←

The external clinician is automatically revoked from her role if consent to referral is withdrawn (A2.2.4, A2.1.12):

(A2.2.4)
isDeactivated(cli, `Ext-treating-clinician`($pat, ra, org, spcty$)) ←
isDeactivated(x, `Consent-to-referral`($pat, ra, org, cli2, spcty$)),
other-referral-consents($0, x, pat, ra, org, cli, spcty$)

(A2.1.12)
other-referral-consents(count$\langle y \rangle, x, pat, ra, org, cli, spcty$) ←
hasActivated(y, `Consent-to-referral`($pat, ra, org, cli, spcty$)),
$x \neq y$

Workgroup Management

As is usual in hospitals, we assume that receptionists register new patients and sign them up for treatment. In contrast to the Spine's policy, Addenbrooke's does not require patients to give explicit consent to treatment. Rather, patient treatment is based on workgroups (§730.13).

We distinguish between two different kinds of workgroups, medical teams and wards. A medical team is a group of clinicians collaboratively treating a patient. A typical team may be headed by a consultant, and supported by specialist registrars, senior house officers, and specialist nurses. Additionally, patients in an in-patient episode are usually treated in a ward. A ward is typically run by a head nurse and a group of other nurses.

Every team is headed by at most one current team member (A3.1.1–7), appointed by a human-resource manager to the role `Head-of-team`(*team*). A similar set of rules governs the appointment, activation and deactivation of `Head-of-ward`(*ward*) roles (A3.4.1–7).

Workgroup membership is managed by human resource managers and workgroup leaders via the registration roles `Register-team-member`(*cli, team, spcty*) and `Register-ward-member`(*cli, ward, spcty*) (A3.2.1-7, A3.5.1-7).

A legitimate relationship exists between a clinician and a patient if the clinician is a member of a workgroup and the patient is currently being treated by that workgroup. Workgroup-based treatment of patients is managed by receptionists (A3.3.1, A3.3.4,

A3.6.1, A3.6.4):

(A3.3.1)
canActivate(rec, `Register-team-episode`(pat, $team$)) ←
    hasActivated(rec, `Receptionist`()),
    canActivate(pat, `Patient`()),
    team-episode-regs(0, pat, $team$)

(A3.3.4)
canDeactivate(rec, x, `Register-team-episode`(pat, $team$)) ←
    hasActivated(rec, `Receptionist`())

(A3.6.1)
canActivate(rec, `Register-ward-episode`(pat, $ward$)) ←
    hasActivated(rec, `Receptionist`()),
    canActivate(pat, `Patient`()),
    ward-episode-regs(0, pat, $ward$)

(A3.6.4)
canDeactivate(rec, x, `Register-ward-episode`(pat, $ward$)) ←
    hasActivated(rec, `Receptionist`())

Team members and heads of wards can also assign patients to teams or wards, respectively:

(A3.3.2)
canActivate(cli, `Register-team-episode`(pat, $team$)) ←
    hasActivated(cli, `Clinician`($spcty$)),
    hasActivated(x, `Register-team-member`(cli, $team$, $spcty$)),
    canActivate(pat, `Patient`()),
    team-episode-regs(0, pat, $team$)

(A3.3.5)
canDeactivate(cli, x, `Register-team-episode`(pat, $team$)) ←
    hasActivated(cli, `Clinician`($spcty$)),
    hasActivated(x, `Register-team-member`(cli, $team$, $spcty$))

(A3.6.2)
canActivate(hd, `Register-ward-episode`(pat, $ward$)) ←
    hasActivated(hd, `Clinician`($spcty$)),
    canActivate(hd, `Head-of-ward`($ward$)),
    canActivate(pat, `Patient`()),
    ward-episode-regs(0, pat, $ward$)

(A3.6.5)
canDeactivate(hd, x, `Register-ward-episode`(pat, $ward$)) ←
    hasActivated(hd, `Clinician`($spcty$)),
    canActivate(hd, `Head-of-ward`($ward$))

Additionally, Caldicott Guardians can cancel workgroup treatment registrations (A3.6.3,

A3.3.3), but cannot sign up patients for treatment:

(A3.6.3)
canDeactivate(cg, x, `Register-ward-episode`($pat, ward$)) ←
    hasActivated(cg, `Caldicott-guardian`())

(A3.3.3)
canDeactivate(cg, x, `Register-team-episode`($pat, team$)) ←
    hasActivated(cg, `Caldicott-guardian`())

The auxiliary role `ADB-treating-clinician`(*pat, group, spcty*) expresses workgroup-based legitimate relationships, based on the clinician being a group member and the patient registered for a team- or ward-episode (A3.8.1–3).

(A3.8.1)
canActivate(cli, `ADB-treating-clinician`($pat, group, spcty$)) ←
    canActivate(cli, `Clinician`($spcty$)),
    hasActivated(x, `Register-team-member`($cli, team, spcty$)),
    hasActivated(y, `Register-team-episode`($pat, team$)),
    $group = team$

(A3.8.2)
canActivate(cli, `ADB-treating-clinician`($pat, group, spcty$)) ←
    canActivate(cli, `Clinician`($spcty$)),
    hasActivated(x, `Register-ward-member`($cli, ward, spcty$)),
    hasActivated(x, `Register-ward-episode`($pat, ward$)),
    $group = ward$

(A3.8.3)
canActivate(cli, `ADB-treating-clinician`($pat, group, spcty$)) ←
    hasActivated(cli, `Emergency-clinician`(pat)),
    $group =$ `A-and-E`,
    $spcty =$ `A-and-E`

This role is used as a prerequisite for adding (A5.1.1) and annotating (A5.1.5) EPR items:

(A5.1.1)
permits(cli, `Add-record-item`(pat)) ←
    hasActivated(cli, `Clinician`($spcty$)),
    canActivate(cli, `ADB-treating-clinician`($pat, group, spcty$))

(A5.1.5)
permits(pat, `Annotate-record-item`(pat, id)) ←
    hasActivated(cli, `Clinician`($spcty$)),
    canActivate(cli, `ADB-treating-clinician`($pat, group, spcty$))

Furthermore, it is a prerequisite for reading EPR items (A5.2.3, A5.3.4, A5.3.8):

(A5.2.3)
$\mathsf{permits}(cli, \texttt{Get-record-item-ids}(pat)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Clinician}(spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{ADB-treating-clinician}(pat, group, spcty))$

(A5.3.4)
$\mathsf{permits}(cli, \texttt{Read-record-item}(pat, id)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Clinician}(spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{ADB-treating-clinician}(pat, group, spcty)),$
$\quad \mathsf{count\text{-}concealed\text{-}by\text{-}patient2}(n, a, b),$
$\quad n = 0,$
$\quad a = (pat, id),$
$\quad b = (\texttt{ADB}, cli, group, spcty),$
$\quad \texttt{Get-record-subjects}(pat, id) \subseteq \texttt{Permitted-subjects}(spcty)$

(A5.3.8)
$\mathsf{permits}(cli, \texttt{Force-read-record-item}(pat, id)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Clinician}(spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{ADB-treating-clinician}(pat, group, spcty))$

All current members of a medical team have permission to read EPR items that have been authored by that team, even if the patient is currently not treated by the team anymore (A5.3.3):

(A5.3.3)
$\mathsf{permits}(cli, \texttt{Read-record-item}(pat, id)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Clinician}(spcty)),$
$\quad \mathsf{hasActivated}(x, \texttt{Register-team-member}(cli, team, spcty)),$
$\quad \texttt{Get-record-group}(pat, id) = team$

Workgroup membership on ADB's system can be used to establish legitimate relationships on the Spine. But as the Spine will contact ADB's RA (RA-ADB) for workgroup credentials (S3.4.1, S3.4.2) (cf. §730.12.0), RA-ADB will in turn request credentials from ADB (R3.1.1, R3.1.2). Therefore, ADB's policy has a canReqCred rule allowing RA-ADB to query the `Register-team-member` and `Register-ward-member` predicates (A1.7.4):

(A1.7.4)
$\mathsf{canReqCred}(x, \texttt{RA-ADB}.\mathsf{hasActivated}(y, \texttt{NHS-health-org-cert}(org, start, end))) \leftarrow$
$\quad org = \texttt{ADB}$

9.4.5 Registration Authorities

RAs are typically local to a particular health organisation but some may also be on a more national level (§730.24.0). We have written a policy for a fictitious RA, RA-ADB, serving Addenbrooke's Foundation Trust and associated hospitals and clinics. As an NHS-approved RA, RA-ADB possesses an RA credential issued by NHS. This credential may be

requested by anyone (R1.2.1):

(R1.2.1)
canReqCred(x, NHS.hasActivated(x, NHS-registration-authority(ra, *start*, *end*))) ←
ra = RA-ADB

RAs issue credentials to professional users for the purpose of managing local workgroups (§730.12.0) and the identification, registration and authentication of role membership (§730.9, §730.21). These access roles are subject to national standards yet to be developed by the NHS (§730.12.2). Our RA policy exemplarily defines user access roles only for clinicians and Caldicott Guardians.

RA credentials are required to be time-limited (§730.24.7). CASSANDRA is flexible enough to encode credentials with validity periods: all RA roles have a start and an end date among their parameters, and the accepting side can specify its own conditions on these dates. For example, it could ignore them, believe them, or impose even stricter freshness conditions (cf. [Riv98]). To authenticate a user's role, the user is issued a credential asserting that someone has activated the corresponding registration role.

Role Credential Management

The only main role defined in RA-ADB's policy is RA-manager(). RA managers sign up professional users for access roles. The RA-manager role itself is a standard delegated registration role: a manager can register a person who has not been so far registered as manager (R1.1.1, R1.1.3)

(R1.1.1)
canActivate(mgr, Register-RA-manager($mgr2$)) ←
hasActivated(mgr, RA-manager()),
ra-manager-regs(0, $mgr2$)

(R1.1.3)
ra-manager-regs(count⟨x⟩, mgr) ←
hasActivated(x, Register-RA-manager(mgr))

This enables that person to activate (R1.1.4) and deactivate (R1.1.5) a manager role:

(R1.1.4)
canActivate(mgr, RA-manager()) ←
hasActivated(x, Register-RA-manager(mgr))

(R1.1.5)
canDeactivate(mgr, mgr, RA-manager()) ←

The role is automatically revoked (R1.1.6) if the registration is cancelled by an RA manager (R1.1.2):

(R1.1.6)
isDeactivated(mgr, RA-manager()) ←
isDeactivated(x, Register-RA-manager(mgr))

(R1.1.2)
canDeactivate(mgr, x, Register-RA-manager($mgr2$)) ←
    hasActivated(mgr, RA-manager())

To register a person as a certified clinician, an RA manager enters the NHS clinician certification role with parameters identifying the clinician, her health organisation, her specialty and a validity period (R2.1.1):

(R2.1.1)
canActivate(mgr, NHS-clinician-cert($org, cli, spcty, start, end$)) ←
    hasActivated(mgr, RA-manager()),
    hasActivated(y, NHS-health-org-cert($org, start2, end2$)),
    $start \in [start2, end2]$,
    $end \in [start2, end2]$,
    $start < end$

The health organisation must be registered on the same RA, and furthermore, the validity period of its registration must contain that of the clinician. Administrators can grant-independently revoke certifications (R2.1.2):

(R2.1.2)
canDeactivate(mgr, x, NHS-clinician-cert($org, cli, spcty, start, end$)) ←
    hasActivated(mgr, RA-manager())

A clinician certification is automatically cancelled if the clinician's health organisation loses all certifications that are valid within the clinician's validity period (R2.1.3, R2.3.3):

(R2.1.3)
isDeactivated(mgr, NHS-clinician-cert($org, cli, spcty, start, end$)) ←
    isDeactivated(x, NHS-health-org-cert($org, start2, end2$)),
    other-NHS-health-org-regs($0, x, org, start2, end2$),
    $start \in [start2, end2]$,
    $end \in [start2, end2]$,
    $start < end$

(R2.3.3)
other-NHS-health-org-regs(count$\langle y \rangle, x, org, start, end$) ←
    hasActivated(y, NHS-health-org-cert($org, start2, end2$)),
    $start \in [start2, end2]$,
    $end \in [start2, end2]$,
    $start < end$,
    $x \neq y \vee start \neq start2 \vee end \neq end2$

The clinician herself, her health organisation, other RAs and the Spine are allowed to request the clinician's credential (R2.1.4–6, R1.2.2–3):

(R2.1.4)
canReqCred(org, RA-ADB.hasActivated(x,
        NHS-clinician-cert($org, cli, spcty, start, end$))) ←
    hasActivated(y, NHS-health-org-cert($org, start2, end2$)),
    Current-time() $\in [start2, end2]$

(R2.1.5)
canReqCred(e, RA-ADB.hasActivated(x,
NHS-clinician-cert($org, cli, spcty, start, end$))) ←
canActivate(e, NHS-service())

(R2.1.6)
canReqCred(cli, RA-ADB.hasActivated(x,
NHS-clinician-cert($org, cli, spcty, start, end$))) ←

(R1.2.2)
canActivate(srv, NHS-service()) ←
canActivate(srv, Registration-authority())

(R1.2.3)
canActivate(srv, NHS-service()) ←
srv = Spine

A similar set of policy rules exist for the certification of Caldicott Guardians (R2.2.1–6) and NHS health organisations (R2.3.1–9).

Workgroup Credential Management

RA-ADB manages workgroup credentials for its registered health organisations, for example for ADB. The Spine can request a Workgroup-member credential certifying a clinician's membership in an organisation's team or a ward (R3.1.3):

(R3.1.3)
canReqCred(Spine, RA-ADB.canActivate(cli,
Workgroup-member($org, group, spcty$))) ←

Membership is deduced by first checking whether the organisation is registered at RA-ADB and then requesting a Register-team-member or a Register-ward-member credential from the organisation (R3.1.1, R3.1.2):

(R3.1.1)
canActivate(cli, Workgroup-member($org, group, spcty$)) ←
hasActivated(x, NHS-health-org-cert($org, start, end$)),
$org \Diamond org$.hasActivated(x, Register-team-member($cli, group, spcty$)),
Current-time() $\in [start, end]$

(R3.1.2)
canActivate(cli, Workgroup-member($org, group, spcty$)) ←
hasActivated(x, NHS-health-org-cert($org, start, end$)),
$org \Diamond org$.hasActivated(x, Register-ward-member($cli, group, spcty$)),
Current-time() $\in [start, end]$

10 Implementation

We have implemented a (so far incomplete) prototype of CASSANDRA, written in Objective Caml (OCaml), a dialect of the ML programming language. We chose OCaml because it is strongly typed, portable, has a module system and automatic memory management. Moreover, its algebraic data types and pattern matching constructs make it especially well suited for implementing formal languages. We did not use OCaml's object-oriented features, and most of the code is functional except for the policy evaluator and the access control engine, both of which also contain imperative code.

At the time of writing, role deactivation, credential requests and the static groundness analyser are still in the process of being implemented. Furthermore, the current prototype only simulates the distributed system, and issued credentials are implemented without encryption and public key signatures. Of the constraint domains presented in Chapter 5, we have implemented the minimal constraint domain $\mathcal{C}_{eq}$ and the one used for our case study, $\mathcal{C}_0$.

The code currently comprises about 6000 lines in total, excluding the code generated by `ocamllex` and `ocamlyacc` for lexing and parsing. More than half of the code is taken up by the module for $\mathcal{C}_0$. The minimal constraint domain $\mathcal{C}_{eq}$ only occupies 250 lines of code, the policy evaluator 1000 lines and the access control engine with the interface about 500 lines.

10.1 Modules

OCaml's module system allows organising modules hierarchically and parameterising a module over other modules. The implementation consists of separate OCaml modules corresponding to the components depicted in Figure 3.1. In particular, the policy evaluator and the policy data structure are both parameterised on the constraint domain. Furthermore, constraint domain modules are written as implementations of an abstract constraint domain definition, so they can be conveniently plugged into the policy evaluator.

10.1.1 Constraint Domains

An OCaml *signature* (an abstract interface) is defined for constraint domains to which concrete implementations must adhere. The signature specifies which functions must be supported and be accessible from the outside:

- `rename x y c` returns the constraint `c` with the variable `x` renamed to `y`. The substitution is capture avoiding.
- `conj c1 c2` returns the conjunction of the constraints `c1` and `c2`.
- `sat c` returns true iff the constraint `c` is satisfiable.
- `impl c1 c2` returns true iff `c1` $\Rightarrow^C$ `c2`. This provides the computable, approximate subsumption operation.
- `elim [x1;...;xn] c` returns a list of quantifier-free and disjunctive-free constraints with the variables `x1`, ..., `xn` existentially eliminated from the constraint `c`.
- `print c` pretty-prints a constraint `c`.
- `parse env s` parses the string `s` with the typing environment `env` and returns a constraint `c`.

We have produced implementations for the constraint domains $\mathcal{C}_{eq}$ and $\mathcal{C}_0$. The implementation of $\mathcal{C}_{eq}$ is straightforward as the associated algorithms are very simple, but the module for $\mathcal{C}_0$ is considerably more involved. It provides automatic type inference that allows us to omit explicit variable typing in constraints, and involves simplification of set expressions, evaluation of functions, and all the constraint graph algorithms for solving inequality and arithmetic constraints described in §5.2.3.

10.1.2 Policy Evaluator

The policy evaluator is implemented as an OCaml functor which allows a module to be parameterised on other modules. The module `Evaluator` is parameterised on the constraint domain interface described above:

```
module Evaluator =
  functor (C: ConstraintType.CONSTRAINT) ->
    struct
    ...
```

It can then be instantiated by a constraint domain module, e.g.

```
module Ev = Eval.Evaluator(C0)
```

The policy evaluator module is an implementation of the SLG^C evaluation algorithm from Chapter 6.

The function call `eval(p,c,policy)` returns the answer to the query `p` $\leftarrow$ `c` with respect to the local policy `policy`. In the current version, remote predicates are treated as local; the most important step in completing the prototype would be to make the policy evaluator truly distributed.

10.1.3 Access Control Engine and Interface

The prototype is operated via a simple text-based interface with which policies can be read in from text files. It also lets users issue service requests and view debugging information (e.g. the current state or the SLG^C proof forest).

The access control engine currently only supports requests to perform an action and to activate a role. It constructs the appropriate policy queries, interprets the answers, and make the necessary changes to the policy, as described in Chapter 7.

10.1.4 Preliminary Results

The prototype was tested with our EHR policy rules by going through various scenarios. The initial test results were promising: all requests were handled within fractions of a second. We believe it would be feasible to use CASSANDRA to enforce our EHR policy on a nation-wide system, despite its relative complexity. Of course, authoritative results can only be produced after completion of a complete and less naive implementation, and under more realistic settings. We have, for example, so far only tested the system with up to 10,000 patients.

Even though queries in $\mathcal{C}_0$ may theoretically be intractable, the test results with the EHR policy suggest that the worst-case does not occur in practice: the policy seems complex but a closer analysis reveals that recursion is very shallow and that nearly all variables become ground at an early stage. This means that neither the evaluation engine nor the constraint solving procedures need to work very hard.

The current implementation is rather inefficient in that credential rules for role activations are stored in a linear list. The lookup of credentials is expensive on a system such as the Spine, with 50 million patients. If an indexed relational database were used instead, the cost of credential lookup would be nearly constant.

Our experiments have highlighted another requirement for policy-based trust management systems that neither our nor existing systems currently fulfil. Human users expect textual justifications of access control decisions, especially if their request is denied; they feel rather frustrated and helpless if the answer is simply *"request denied"*, especially if the policy is complex or unknown to the user. Such explanations could be collected from annotations of policy rules used during deduction. The problem is non-trivial as deduction proofs can be long and access denials can have many and far-reaching reasons. More worryingly, the textual justification may reveal more (and perhaps, sensitive) information than could have been deduced from the fact of request denial alone: consider, for example, a response such as *"access denied because your daughter has prohibited you from accessing all her records with the subject 'abortion'"*.

11 Discussion and Conclusions

In the last decade, the Internet has come to play an increasingly important role in everyday life in our society. This trend will continue with the emergence of new widely-distributed applications. In the commercial sector, interoperable Web Services offer business facilities over the World Wide Web. Pervasive environments enable people, or devices on their behalf, to communicate and collaborate wirelessly with peers in a highly dynamic network with changing contexts. In the public sector, distributed services are being developed for online health care, police records, voting and taxes. As we become more and more dependent on such applications simplifying and automating day-to-day tasks, it is of utmost importance to develop means to guarantee security of their operations, especially with regards to privacy and confidentiality of data. The security requirements for widely-distributed and dynamic architectures are highly challenging, and it is essential to specify their security requirements in a high-level policy language that is flexible, expressive, widely applicable and efficient.

In this thesis we proposed a flexible trust management system, CASSANDRA, with a general-purpose policy language that is expressive enough to capture the authorisation requirements of the proposed NHS data spine, the largest IT project in the UK. This chapter discusses CASSANDRA's features in comparison to other trust management systems, and summarises the lessons learnt from the case study (§11.1). We recapitulate our main contributions in §11.2, and outline our intended future work in §11.3.

11.1 Discussion

11.1.1 Expressive Power

A general-purpose policy language has to be flexible enough to be able to express the range of policy idioms needed in real-world applications. Many existing languages have special constructs for each policy idiom: for example, most RBAC models directly support some

variant of role hierarchy, some support separation of duties constraints, SPKI/SDSI [Ell99, EFL+99] has a flag for permission delegation, and OASIS [HBM98, YMB02, BMY02] defines certificates for role appointment. If a language is not expressive enough it is often extended when the requirements change. Language extensions also entail a change in the definition of the language semantics.

For example, in the RT framework [LMW02], there are separate language extensions for the support of parameterised roles, for grouping resources, for expressing threshold and separation of duties, for delegation of role activations, and for constraining role parameters [LM03]. Each of these extensions comes with a new semantics. However, defining a completely new language semantics for each new extension is tedious and makes it hard to compare it to the various sublanguages.

CASSANDRA's policy language is unique in that its expressiveness is parameterised. The base language is designed to be simple, small and very general. The actual expressiveness is induced by the constraint domain that can be chosen according to need. Since the definition of the language semantics is independent of the chosen constraint domain, it can be seen as a plug-in module, on the level of the formal semantics as well as on the implementation level.

It is essentially the generality of the base language combined with the freedom to choose an appropriate constraint domain that makes CASSANDRA so expressive and flexible. However, there are a number of other language features that enable CASSANDRA to express policies as complex as in our EHR case study.

One of the most prominent features in CASSANDRA is the tagging of predicates with location and issuer. Recall that the location indicates where a predicate should be deduced, and the issuer states who vouches for the truth of the predicate, or who has authority to define that predicate. The issuer prefix was inspired by the notion of localised name space in SPKI/SDSI, and by the similar prefix constructions in RT, QCM [GJ00b] and SD3 [Jim01]. SD3 also has a location prefix, but there it has the simpler meaning of being the IP address of the issuer. In contrast, the location and the issuer in CASSANDRA are separate and independent. The full flexibility of this feature is required, for example, to retrieve credentials of the form *"x vouches for the fact that y has activated a role with parameters containing z"*, where the credential is located neither at x nor at y but rather in z's policy. An example of such a credential is the `NHS-health-org-cert` credential from our case study. Consider for example the Spine rule that allows certified NHS health organisations to request an agent credential (S1.4.8):

$$\begin{aligned}&\mathsf{canReqCred}(org, \texttt{Spine}.\mathsf{canActivate}(ag, \texttt{Agent}(pat))) \leftarrow \\ &\quad org\Diamond ra.\mathsf{hasActivated}(x, \texttt{NHS-health-org-cert}(org, start, end)), \\ &\quad \mathsf{canActivate}(ra, \texttt{Registration-authority}()), \\ &\quad \texttt{Current-time}() \in [start, end]\end{aligned}$$

As can be seen from the first body predicate, the credential is signed and vouched for by an RA and states that some x (this will be an RA manager) has activated a role indicating that *org* is an accredited NHS health organisation. The rule expects this credential to be stored neither at the RA (the issuer) nor with the RA manager (the subject), but in the health organisation's (*org*) policy. Most trust management systems assume that credentials are always stored with the issuer. RT is a bit more flexible in that it allows credentials also to be stored with the subject of the credential. In CASSANDRA, credentials can be stored and retrieved from everywhere.

Another important, but rather subtle feature is the explicit subject parameter in the six special predicates. For example, in the predicate hasActivated, the first parameter indicates *who* has activated the role. The explicit parameter allows us to specify a different subject for the head and for each body predicate. We can thus express rules of the form *"x has property X if y has property Y and z has property Z and ..."*. This is also possible in Binder [DeT02], but in some other languages with designated authorisation predicates, e.g. RT or Oasis, the subject is implicitly the same for the head and the entire body. In such languages one can only express *"x has property X if x has properties Y and Z"*. This feature is crucial for encoding many policy idioms directly in the language. For example, role appointment rules are inherently of a form where the subjects of the goal and the conditions are different: privileges are granted to a person if *another* person has appointed them to do so.

Aggregation rules have been included in CASSANDRA's design as a generalisation of predicate negation and have proven highly valuable in our case study. Most policy languages do not allow negation as it easily leads to intractability, semantic ambiguity, and undecidability. Lithium [HW03] is an example of a policy language allowing negated predicates. However, with negated predicates, the kind of negation we are most interested in cannot be expressed: universally quantified negated statements such as *"nobody has activated role R"*. In contrast, a negated predicate $\neg\mathsf{hasActivated}(x, \mathtt{R}())$ only says that there exists an x that has not activated the role.

In Lithium, negated permission predicates can even be in the head of a rule. This feature is used to express explicit prohibition, much like the "negative authorisations" in Ponder [DDLS01, Dam02]. The design of CASSANDRA does not include prohibitions, as we believe that it is sufficient to prohibit everything that is not explicitly allowed. With this assumption, we avoid having to deal with the problem of conflict resolution between prohibitions and permissions. Moreover, the formal framework allows us to prove meta-theorems about a policy, for example that an entity *cannot* perform some action. Meta-proofs could perhaps even be machine-checked; this is part of our intended future work.

Aggregation is also used for many other purposes. It can generally be used for universally quantified statements, for example for the condition that all affected third parties have given explicit consent to disclosure of a record item (Rules S2.2.17 and S5.3.1):

$$\begin{aligned}&\mathsf{third\text{-}party\text{-}consent}(\mathsf{group}\langle \mathit{consenter}\rangle, \mathit{pat}, \mathit{id}) \leftarrow\\ &\quad \mathsf{hasActivated}(x, \mathtt{Third\text{-}party\text{-}consent}(\mathit{consenter}, \mathit{pat}, \mathit{id}))\end{aligned}$$

$$\begin{aligned}&\mathsf{permits}(\mathit{pat}, \mathsf{Read\text{-}spine\text{-}record\text{-}item}(\mathit{pat}, \mathit{id})) \leftarrow\\ &\quad \mathsf{hasActivated}(\mathit{pat}, \mathtt{Patient}()),\\ &\quad \mathsf{hasActivated}(\mathit{pat}, \mathtt{One\text{-}off\text{-}consent}(\mathit{pat})),\\ &\quad \mathsf{count\text{-}concealed\text{-}by\text{-}spine\text{-}patient}(0, (\mathit{pat}, \mathit{id}), \mathit{spec}),\\ &\quad \mathsf{count\text{-}concealed\text{-}by\text{-}spine\text{-}clinician}(0, \mathit{pat}, \mathit{id}),\\ &\quad \mathsf{third\text{-}party\text{-}consent}(\mathit{consenters}, \mathit{pat}, \mathit{id}),\\ &\quad \mathit{spec} = (\mathtt{No\text{-}org}, \mathtt{pat}, \mathtt{No\text{-}group}, \mathtt{No\text{-}spcty}),\\ &\quad \mathtt{Get\text{-}spine\text{-}record\text{-}third\text{-}parties}(\mathit{pat}, \mathit{id}) \subseteq \mathit{consenters}\end{aligned}$$

Aggregation is also useful in rules with cardinality constraints. For example, agent registration is an example of role appointment combined with a cardinality constraint (Rules S1.4.14, S1.4.9):

$$\begin{aligned}&\mathsf{agent\text{-}regs}(\mathsf{count}\langle x\rangle, \mathit{pat}) \leftarrow\\ &\quad \mathsf{hasActivated}(\mathit{pat}, \mathtt{Register\text{-}agent}(x, \mathit{pat}))\end{aligned}$$

$$\mathsf{canActivate}(pat, \texttt{Register-agent}(agent, pat)) \leftarrow$$
$$\quad \mathsf{hasActivated}(pat, \texttt{Patient}()),$$
$$\quad \mathsf{agent\text{-}regs}(n, pat),$$
$$\quad n < 3$$

Another use of aggregation is in uniqueness constraints. These have so far not been considered in the literature: roles that can be activated if nobody else has activated them. Closely related to uniqueness constraints is a variant of appointment revocation in a context where an appointee can have more than one appointer and revocation should only be performed if all appointments are cancelled. For example, agents can be registered by both patients and their GPs, and only if all registrations are cancelled, an active agent role is deactivated (Rules S1.4.4 and S1.4.3):

$$\mathsf{other\text{-}agent\text{-}regs}(\mathsf{count}\langle y\rangle, x, ag, pat) \leftarrow$$
$$\quad \mathsf{hasActivated}(y, \texttt{Register-agent}(ag, pat)),$$
$$\quad x \neq y$$

$$\mathsf{isDeactivated}(ag, \texttt{Agent}(pat)) \leftarrow$$
$$\quad \mathsf{isDeactivated}(x, \texttt{Register-agent}(ag, pat)),$$
$$\quad \mathsf{other\text{-}agent\text{-}regs}(0, x, ag, pat)$$

11.1.2 Distributed Credential Management

All trust management systems provide authorisations based on credentials. However, they vary greatly in the versatility of their credential management mechanisms. Some trust management systems such as PolicyMaker [BFL96, BFK99c], KeyNote [BFK99a, BFK99b] and SPKI/SDSI do not support automated credential retrieval. In these systems it is the responsibility of the user to submit the right credentials.

RT, QCM and SD3 all have mechanisms for automated credential retrieval. Roles in RT (or set expressions in QCM, and predicates in SD3) can be prefixed by a name to indicate that they correspond to a foreign credential to be retrieved over the network. As discussed before, these systems do not specify an independent credential location and hence cannot retrieve credentials that are not stored in the issuer's (or, in the case of RT, in the role subject's) policy. Another consequence of this limitation is that foreign credentials are always requested over the network (unless the retrieval mechanism has been turned off altogether, in the case of QCM and SD3) whenever they have not been submitted. In CASSANDRA, setting the location prefix of a predicate to the location of the rule forces the query evaluation engine to find the corresponding credential locally, i.e. amongst the credential rules in the policy or the credential submitted along with the request. The location prefix provides a very fine granularity of control over automated credential retrieval.

Credentials may be confidential and should, just like other resources, be protected by policy-based access control. CASSANDRA's integrated trust negotiation mechanism is another important unique feature. Just as actions are protected by permits policy rules, credentials, themselves being part of a policy, are protected by canReqCred policy rules.

In CASSANDRA, the credential's parameters in the request are allowed to be not fully instantiated. The credential that is actually sent back to the requester is, in general, fully instantiated and can thus convey new information to the requester. A credential request can thus simulate database queries. The combination of location and issuer prefixes, canReqCred rules, and the definition of the reqCred transition in the operational semantics

forms a flexible mechanism that unifies trust management, credential retrieval and trust negotiation.

11.1.3 Scalability and Complexity

A trust management system should be scalable: it should be easy to write and administer policies, and requests should be decided efficiently, even in the presence of a complex policy and a large number of entities.

Roles are a means of grouping entities, thereby reducing the complexity of policy administration and facilitating more concise policies, especially with parameterised roles, than in systems that are not role based, e.g. PolicyMaker or KeyNote. SD3 does not have roles either, as it is designed to be a very general language without any inherent access control meaning, but it would probably be not too hard to extend it to support parameterised roles. OASIS and RT also support parameterised roles, and the name space specifications in SPKI/SDSI could be interpreted as non-parameterised roles. However, it turned out that the most important benefit of roles in CASSANDRA was not the scalability issue. Rather, roles in combination with the access control engine enable us to express state-dependent policies, by viewing a role as an attribute that can be set by activating it and unset by deactivating it. In our case study, we use such roles to express relevant state changes. For example, a patient is registered by somebody activating the `Register-patient` role; or, a patient denies access to a record item by activating an appropriate access denial role.

The efficiency of query evaluation in CASSANDRA depends mainly on the chosen constraint domain. With a very simple constraint domain such as $\mathcal{C}_{eq}$ or $\mathcal{C}_{tup}$, evaluation is tractable (PTIME) for all policies. With a highly expressive constraint domain such as $\mathcal{C}_{\neq,<}$ and $\mathcal{C}_0$ (restricted with groundness analysis), the worst-case complexity is DEXPTIME-complete [Rev95]. However, the worst-case seems to occur only in pathological cases (such as using set constraints to encode the Hamiltonian Cycle problem); we could not come up with a realistic policy that exhibits exponential behaviour. In particular, query evaluation of our EHR policy is simple as the recursion depth is low (so the proof forest will be small) and, most importantly, variables get instantiated at a very early stage, also due to the groundness restrictions.

KeyNote, RT, Lithium, QCM and SD3 are all tractable languages, at the expense of limited expressiveness as they are based on simple Datalog without constraints, or on a heavily restricted form of first order logic, in the case of Lithium. $RT_1^{\mathcal{C}}$ [LM03] is based on Datalog$^{\mathcal{C}}$, but the use of constraints is heavily restricted to keep the language tractable; in essence, constraints are only used to restrict the range of a single variable. PolicyMaker, OASIS and Ponder are amongst those languages that allow policies to be undecidable.

11.1.4 Case Study: Lessons Learnt

The complexity and constantly evolving nature of the Spine's security and confidentiality requirements necessitate the use of a policy language in order to separate policy from implementation. The policy language must be efficiently machine-enforceable; it must be high-level and sufficiently simple so the policy can be easily modified and read; and it must be expressive and flexible in order to accommodate for current and unforeseeable future requirements.

We have presented a complete CASSANDRA policy governing access to health records, based on official NHS and DoH documents. The case study shows that CASSANDRA is

sufficiently expressive for NPfIT and other large-scale real-world applications with highly challenging security requirements. Our preliminary experiments with the policy running on our prototype implementation of CASSANDRA strongly suggest that the system will also be efficient in practice.

The detailed description of the policy rules could be seen as a translation of the formal CASSANDRA rules back into plain English. It is conceivable that the translation process into a subset of English could be automated. We are currently communicating with the NHS to see whether our description really matches their requirements and intentions. If approved by the NHS, such a detailed and semi-formal description would be useful for the NHS's IT suppliers, who have only been given the rather sketchy OBS. Also, the description could be given to the public to put the NHS's authorisation policy under public and legal expert scrutiny and could help answering the question whether the proposed Spine will fulfil all legal and ethical confidentiality requirements. In the best case, it could calm the public's unease and relieve the current uncertainty about the project.

One of the lessons learnt from the case study is that the hardest part about writing policy is not the translation into a formal language, but rather understanding the intended requirements. As expected, the available specification documents are unclear, ambiguous, and – above all – incomplete, rather than contradictory. Certainly, many of the gaps could be filled in with common sense. Still, as mentioned above, it will be important to get some official feedback. However, once the requirements are understood and complete, the translation process is relatively straightforward: most of the informal, "intuitive" requirements statements are already approximately of the form *"if ⟨condition⟩ then ⟨goal⟩"*.

An obvious question to ask is: is the formal policy correct and does it do what it is supposed to do? Since translating the informal rules into CASSANDRA was rather straightforward, we believe that the primary source of "incorrectness" would be the requirements in the first place. In such a large and intricate system, it is difficult to fully understand the implications of the requirements. However, having translated the requirements into formal CASSANDRA policy rules is the first step towards proving meta-level correctness properties; for example, that users can really only log on with exactly one main role at any time.

The case study also provided many valuable lessons on more technical aspects. Our case study exhibits interesting variants and combinations of policy idioms; the idiom of role appointment was needed especially frequently, as well as separation of duties, uniqueness and cardinality constraints, and automated trust negotiation. Certainly, CASSANDRA's flexibility and expressiveness paid off well, as we could express all these variants directly in the language without having to extend it with special constructs. The design of the constraint domain $\mathcal{C}_0$ was mainly guided by the EHR policy's requirements. The case study also highlighted important features of CASSANDRA that make it unique, e.g. predicates with an issuer and a location (for credentials that are neither stored with the issuer nor with the subject of the predicate); aggregation operators (for universally quantified negation, and cardinality and uniqueness constraints), and an explicit subject in all special predicates.

Our case study has significant implications for the research area of trust management as a whole. Most other systems have only been applied to relatively simple applications or academic toy examples. There has been a major lack of real-world policy examples that are both large and complex — our EHR policy example fills this gap. It is a strong counterexample to the claim that real-world applications do not need expressive policy languages. Indeed, it is hard to imagine how the ambitious NHS project could be realised successfully without a flexible, distributed access control system that allows the authorisation policy to be modified easily.

Our policy can be used as a benchmark for existing and future policy languages, as a guideline for language design and as a tool for the difficult task of comparing different policy languages. It would be rewarding to translate the policy into another language, and to analyse which constructs cannot be translated, and which features can perhaps be more easily expressed in a different language.

11.2 Summary of Contributions

The main contribution of this dissertation is the proposal of a policy-based trust management system with a high grade of flexibility, CASSANDRA, and the large-scale case study on an authorisation policy for a national EHR system. In more detail, the following contributions have been made:

- Proposing Datalog$^{\mathcal{C}}$ as the basis of a policy language to parameterise expressiveness.
- Extending Datalog$^{\mathcal{C}}$ with parameterised roles and actions, access control predicates, signed credentials with network locations, and aggregation operators.
- Providing a formal semantics of the policy language based on fixed points.
- Designing a useful hierarchy of constraint domains for policy specification, along with type systems and constraint solving algorithms for constraint satisfaction checking, subsumption checking and existential quantifier elimination.
- Proving constraint compactness of constraint domains, a formal condition to ensure finiteness of the fixed point semantics, regardless of the policy and the query.
- Developing a distributed top-down query evaluation algorithm based on SLG$^{\mathcal{C}}$ resolution that terminates whenever the fixed point semantics is finite, and in particular, whenever the constraint domain is constraint compact.
- Proposing groundness analysis as a means to sanity-check policies, lift syntactic restrictions on aggregation rules, and to restrict overly expressive constraint domains to a constraint compact fragment during run-time.
- Developing a procedure to analyse groundness of variables in policies based on SLG$^{\mathcal{C}}$ resolution, and formally proving its correctness.
- Specifying an operational semantics of the access control engine by a labelled transition system, with rules for performing an action, activating a role, deactivating a role, and requesting a credential.
- Illustrating the system's versatility by showing encodings of various policy idioms, including separation of duties, role hierarchy, role delegation and role appointment.
- Providing the research area of trust management with the as yet largest and most complex real-world case study on authorisation policy: a complete, distributed policy for the national EHR system proposed by the NHS. The case study confirms that current and future applications will indeed require policy-based trust management with a high degree of flexibility and expressive power.
- Implementing a (still incomplete) prototype of the system, and testing it with the EHR policy.

11.3 Future Work

We conclude this dissertation with an outline of potential future work. We concentrate on four main areas in which we envisage to do further work: proving meta-properties, formal models, implementation, and the case study.

11.3.1 Proofs

One of the big advantages of having an explicit and formal policy is that it can be formally analysed. Policy analysis so far has focussed mainly on the detection of rule conflicts and inconsistencies [BLR03, JSS97]. To unleash the full potential of policy specification, we also need methodologies and tools for analysing more complex security properties and implications of policies. Some research has already been done on this topic, based on the *RT* language [LT04], but a lot more needs to be done on more expressive languages such as CASSANDRA.

In applications secured by "low-level" access control mechanisms such as DAC, MAC or RBAC, the security goals are only implicit and often hard-coded. Proving conformance to high-level policies in such systems is non-trivial. Policy specification makes the security goals explicit.

CASSANDRA was designed to satisfy complex policy requirements and at the same time be simple enough that its language and access control semantics can be formally specified. Although policy specification makes high-level security goals explicit, there will always remain higher-level "meta-properties" that cannot be stated directly. For example, it is not possible to express prohibition in CASSANDRA directly, so it would be useful to be able to prove that some entity will never be able to perform some action. We plan to use our formal framework to prove interesting meta-properties about specific policies or policy idioms. Proving that the encoding of a particular policy idiom is correct will be straightforward in many cases, as the encoding is often a very direct translation of the policy idiom's intended meaning. We expect that such simple proofs will share many common features. By studying their structure and logic, we wish to find a way to express the proofs in a form that makes them amenable to machine-checking. It is also conceivable that a simple family of properties can be proved mechanically by theorem provers such as HOL [GM93], Coq [BBC+97] or Isabelle [Pau94]. Type systems have also been proposed for statically verifying implementations of Datalog-based policies [FGM05].

A local policy can refer to remote policies; in this context, it would be useful to be able to prove assertions such as *"X will never gain permission to read record item Y under this local policy, no matter in which way the remote policy of Z changes"*, or *"the local policy relies on the remote policy of Z, but even if Z is malicious and does not act according to the semantics, the worst thing that could happen is E"*. These kinds of propositions are interesting because entities have complete knowledge only of their own policy, and yet, to some extent, rely on other policies they partially trust but cannot control. They cannot completely exclude the possibility that the trusted party behaves unexpectedly. This, of course, brings with it some risk, and it is important to be able to formally examine the extent of this risk.

A third class of useful properties to prove are meta-theoretic results about the limitations of expressiveness. By proving statements such as *"policy idiom X cannot be expressed with constraint domain Y"* or *"policy idiom X cannot be expressed without aggregation"*, we could gain a better understanding of the basic building blocks of policy idioms.

11.3.2 Formal Models

We have formally specified CASSANDRA's language semantics as well as the operational semantics of the policy evaluation engine; together, they form a complete formal model of the system. However, real implementations would have to deal with many issues that are missing in the high-level model. It would therefore be desirable to have a low-level model of the system that specifies the underlying communication protocols, the public-key infrastructure and the design of credentials. The low-level model would also specify the behaviour of the system in the presence of network failure or transmission delays. The model could be formalised using techniques from standard transition-based operational semantics and the Spi calculus [AG99], an extension of the π-calculus with cryptographic primitives.

This future task would also involve proving theorems about the correspondence between the high-level and the low-level models to show that the latter is a correct implementation of the former. This will also increase confidence in the correctness of actual implementations.

The model also needs to be extended to be able to deal with policy changes. At the moment, the model assumes that policies stay the same, apart from the automatic changes to reflect the state of the system, i.e., those due to role activations/deactivations and credential requests. In reality, of course, policies will frequently also be updated manually by policy writers. As we are considering distributed policies, such updates will have propagating and, perhaps, unexpected and inconsistent effects. There has already been some work done on OASIS policy evolution that is consistent and compliant with some meta-policy [Bel04] that might be used as a starting point.

11.3.3 Implementation

Our current implementation of CASSANDRA is still incomplete. We need to add the functionalities of role deactivation and credential requests, and implement the static groundness analyser. Also, at the moment, all distributed features are merely simulated. To gather more reliable test results, we plan to build a more realistic, truly distributed prototype. In particular, network communication will be encrypted, the policy language will be encoded in XML, and the system will be embedded in a PKI with credentials encoded as X.509 certificates. User interfaces need to be devised that facilitate user-friendly policy authoring and management.

We also plan to investigate optimisation strategies for making the implementation efficient. For a policy with millions of users as is the case in our NHS example, the bottleneck lies in the lookup of parameterised role activations. A promising optimisation approach that we will explore is to devise a suitable indexing scheme for such activations and to store them in a relational database.

It would be interesting to see whether it is feasible to implement a simple proof checker to verify every answer from query evaluation, as proposed in [Jim01]. This could significantly reduce the size of the trusted computing base and increase our confidence in the correctness of the implementation.

11.3.4 Case Studies

More policy case studies are needed. A promising application area is the emerging field of Semantic Web Services [FHLW02] — an area with the potential to revolutionise e-Commerce. Semantic Web Services are a combination of web services and the Semantic Web: autonomous heterogeneous, pervasive web-based software components that support

automated and unanticipated service discovery, invocation, composition and interoperability through machine-understandable semantic descriptions. As such, distributed trust management is ideally suited for securing future Semantic Web Services [FJ02, ASW04]. It would be necessary to study how to interface the policy language with Semantic Web technologies, in particular the Resource Description Framework (RDF), the Ontology Web Language (OWL) and the emerging DAML-S/OWL-S ontologies. Then, for a sample of interoperating Semantic Web Services, authorisation policies could be constructed in a similar manner as for our NHS example.

With the NHS case study, we have produced a large and complex authorisation policy example. However, it is based on informal documents that are out-of-date, incomplete, and, in many places, subject to ongoing debate. In order for the policy to become an even more convincing benchmark example for future language design and comparisons, we need to refine it and verify its correctness by further discussing these issues with the NHS. This is a difficult and time-consuming task, one reason being that the application domain experts and stakeholders are not trust management experts, in general, and cannot be expected to read and understand the full formal policy.

We hope to continue our collaboration with the NHS. NPfIT representatives have agreed that their current informal specification process is somewhat inadequate, and have expressed interest in developing a formal CASSANDRA policy for still contended areas of information governance. In particular, we will continue updating the policy fragment on Sealed Envelopes, based on results from ongoing NPfIT consultations (e.g. [Osw05]), and point out ambiguities and and other weaknesses to NPfIT.

From our discussions with the NHS so far we have learnt that the best way to disseminate trust management technology is to provide user-friendly demonstrator applications with which stakeholders — representatives from the NHS and their IT suppliers, health professionals and patients — can empirically verify the consequences of the policy.

The NPfIT project currently has a huge image problem, especially due to the perceived dangers to patient confidentiality. Perhaps we could help the NHS towards designing a system that would be more trusted by those who are supposed to use it — a system that is flexible, feasible, and meets even the highest confidentiality requirements. At stake are human lives and huge financial resources.

A

Policy rules for NHS electronic health record system

The following is a complete list of all CASSANDRA rules for our EHR case study. Appendix A.1 contains the policy rules for the Spine, A.2 for the PDS, A.3 for Addenbrooke's Hospital (ADB), and A.4 contains the rules for Addenbrooke's RA (RA-ADB).

A.1 Policy for the Spine

A.1.1 Main access roles

Clinician

(S1.1.1)
$$\begin{aligned}&\mathsf{canActivate}(cli, \texttt{Spine-clinician}(ra, org, spcty)) \leftarrow \\ &\quad ra.\mathsf{hasActivated}(x, \texttt{NHS-clinician-cert}(org, cli, spcty, start, end)), \\ &\quad \mathsf{canActivate}(ra, \texttt{Registration-authority}()), \\ &\quad \mathsf{no\text{-}main\text{-}role\text{-}active}(cli), \\ &\quad \texttt{Current-time}() \in [start, end]\end{aligned}$$

(S1.1.2)
$$\begin{aligned}&\mathsf{canActivate}(cli, \texttt{Spine-clinician}(ra, org, spcty)) \leftarrow \\ &\quad ra \Diamond ra.\mathsf{hasActivated}(x, \texttt{NHS-clinician-cert}(org, cli, spcty, start, end)), \\ &\quad \mathsf{canActivate}(ra, \texttt{Registration-authority}()), \\ &\quad \mathsf{no\text{-}main\text{-}role\text{-}active}(cli), \\ &\quad \texttt{Current-time}() \in [start, end]\end{aligned}$$

(S1.1.3)
$$\mathsf{canDeactivate}(cli, cli, \texttt{Spine-clinician}(ra, org, spcty)) \leftarrow$$

(S1.1.4)
count-spine-clinician-activations(count⟨u⟩, $user$) ←
    hasActivated($user$, `Spine-clinician`(ra, org, $spcty$))

Administrator

(S1.2.1)
canActivate(adm, `Spine-admin`()) ←
    hasActivated(x, `Register-spine-admin`(adm)),
    no-main-role-active(adm)

(S1.2.2)
canDeactivate(adm, adm, `Spine-admin`()) ←

(S1.2.3)
isDeactivated(adm, `Spine-admin`()) ←
    isDeactivated(x, `Register-spine-admin`(adm))

(S1.2.4)
count-spine-admin-activations(count⟨u⟩, $user$) ←
    hasActivated($user$, `Spine-admin`())

(S1.2.5)
canActivate(adm, `Register-spine-admin`($adm2$)) ←
    hasActivated(adm, `Spine-admin`()),
    spine-admin-regs(0, $adm2$)

(S1.2.6)
canDeactivate(adm, x, `Register-spine-admin`($adm2$)) ←
    hasActivated(adm, `Spine-admin`())

(S1.2.7)
spine-admin-regs(count⟨x⟩, adm) ←
    hasActivated(x, `Register-spine-admin`(adm))

Patient

(S1.3.1)
canActivate(pat, `Patient`()) ←
    hasActivated(x, `Register-patient`(pat)),
    no-main-role-active(pat),
    `PDS`◊`PDS`.hasActivated(y, `Register-patient`(pat))

(S1.3.2)
canDeactivate(pat, pat, `Patient`()) ←

(S1.3.3)
isDeactivated(pat, `Patient`()) ←
    isDeactivated(x, `Register-patient`(pat))

(S1.3.4)
count-patient-activations(count⟨u⟩, $user$) ←
    hasActivated($user$, `Patient`())

(S1.3.5)
canActivate(adm, `Register-patient`(pat)) ←
    hasActivated(adm, `Spine-admin`()),
    patient-regs(0, pat)

(S1.3.6)
canDeactivate(adm, x, `Register-patient`(pat)) ←
    hasActivated(adm, `Spine-admin`())

(S1.3.7)
patient-regs(count⟨x⟩, pat) ←
    hasActivated(x, `Register-patient`(pat))

Agent

(S1.4.1)
canActivate(ag, `Agent`(pat)) ←
    hasActivated(x, `Register-agent`(ag, pat)),
    `PDS`◊`PDS`.hasActivated(y, `Register-patient`(ag)),
    no-main-role-active(ag)

(S1.4.2)
canDeactivate(ag, ag, `Agent`(pat)) ←

(S1.4.3)
isDeactivated(ag, `Agent`(pat)) ←
    isDeactivated(x, `Register-agent`(ag, pat)),
    other-agent-regs(0, x, ag, pat)

(S1.4.4)
other-agent-regs(count⟨y⟩, x, ag, pat) ←
    hasActivated(y, `Register-agent`(ag, pat)),
    $x \neq y$

(S1.4.5)
count-agent-activations(count⟨u⟩, $user$) ←
    hasActivated($user$, `Agent`(pat))

(S1.4.6)
canReqCred(ag, `Spine`.canActivate(ag, `Agent`(pat))) ←
    hasActivated(ag, `Agent`(pat))

(S1.4.7)
canReqCred(org, `Spine`.canActivate(ag, `Agent`(pat))) ←
    ra.hasActivated(x, `NHS-health-org-cert`(org, $start$, end)),
    canActivate(ra, `Registration-authority`()),
    `Current-time`() ∈ [$start$, end]

(S1.4.8)
canReqCred(org, Spine.canActivate(ag, Agent(pat))) ←
$org\Diamond ra$.hasActivated(x, NHS-health-org-cert(org, $start$, end)),
canActivate(ra, Registration-authority()),
Current-time() ∈ [$start$, end]

(S1.4.9)
canActivate(pat, Register-agent($agent$, pat)) ←
hasActivated(pat, Patient()),
agent-regs(n, pat),
$n < 3$

(S1.4.10)
canActivate(cli, Register-agent($agent$, pat)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
canActivate(cli, General-practitioner(pat))

(S1.4.11)
canDeactivate(pat, pat, Register-agent($agent$, pat)) ←
hasActivated(pat, Patient())

(S1.4.12)
canDeactivate(cli, x, Register-agent($agent$, pat)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
canActivate(cli, General-practitioner(pat))

(S1.4.13)
isDeactivated(x, Register-agent($agent$, pat)) ←
isDeactivated(y, Register-patient(pat))

(S1.4.14)
agent-regs(count⟨x⟩, pat) ←
hasActivated(pat, Register-agent(x, pat))

Other

(S1.5.1)
canActivate(ra, Registration-authority()) ←
NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end)),
Current-time() ∈ [$start$, end]

(S1.5.2)
canActivate(ra, Registration-authority()) ←
$ra\Diamond$NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end)),
Current-time() ∈ [$start$, end]

(S1.5.3)
no-main-role-active($user$) ←
count-agent-activations(n, $user$),
count-spine-clinician-activations(n, $user$),
count-spine-admin-activations(n, $user$),
count-patient-activations(n, $user$),
count-third-party-activations(n, $user$),
$n = 0$

A.1.2 Express consent

One-off consent

(S2.1.1)
canActivate(pat, `One-off-consent`(pat)) ←
hasActivated(pat, `Patient`())

(S2.1.2)
canActivate(ag, `One-off-consent`(pat)) ←
hasActivated(ag, `Agent`(pat))

(S2.1.3)
canActivate(cli, `One-off-consent`(pat)) ←
hasActivated(cli, `Spine-clinician`(ra, org, $spcty$)),
canActivate(cli, `Treating-clinician`(pat, org, $spcty$))

(S2.1.4)
canDeactivate(pat, x, `One-off-consent`(pat)) ←
hasActivated(pat, `Patient`())

(S2.1.5)
canDeactivate(ag, x, `One-off-consent`(pat)) ←
hasActivated(ag, `Agent`(pat))

(S2.1.6)
canDeactivate(cli, x, `One-off-consent`(pat)) ←
hasActivated(cli, `Spine-clinician`(ra, org, $spcty$)),
canActivate(cli, `Treating-clinician`(pat, org, $spcty$))

(S2.1.7)
isDeactivated(x, `One-off-consent`(pat)) ←
isDeactivated(y, `Register-patient`(pat))

Third-party consent

(S2.2.1)
canActivate(pat, `Request-third-party-consent`(x, pat, id)) ←
hasActivated(pat, `Patient`()),
$x \in$ `Get-spine-record-third-parties`(pat, id)

(S2.2.2)
canActivate(ag, `Request-third-party-consent`(x, pat, id)) ←
    hasActivated(ag, `Agent`(pat)),
    $x \in$ `Get-spine-record-third-parties`(pat, id)

(S2.2.3)
canActivate(cli, `Request-third-party-consent`(x, pat, id)) ←
    hasActivated(cli, `Spine-clinician`($ra, org, spcty$)),
    canActivate(cli, `Treating-clinician`($pat, org, spcty$)),
    $x \in$ `Get-spine-record-third-parties`(pat, id)

(S2.2.4)
canDeactivate(pat, y, `Request-third-party-consent`(x, pat, id)) ←
    hasActivated(pat, `Patient`())

(S2.2.5)
canDeactivate(ag, y, `Request-third-party-consent`(x, pat, id)) ←
    hasActivated(pat, `Agent`(pat))

(S2.2.6)
canDeactivate(cli, y, `Request-third-party-consent`(x, pat, id)) ←
    hasActivated(cli, `Spine-clinician`($ra, org, spcty$))

(S2.2.7)
canDeactivate(x, y, `Request-third-party-consent`(x, pat, id)) ←
    hasActivated(x, `Third-party`())

(S2.2.8)
isDeactivated(x, `Request-third-party-consent`(y, pat, id)) ←
    isDeactivated(z, `Register-patient`(pat))

(S2.2.9)
other-third-party-consent-requests(count$\langle x \rangle, y, z$) ←
    hasActivated(x, `Request-third-party-consent`(z, pat, id)),
    $x \neq y$

(S2.2.10)
canActivate(x, `Third-party`()) ←
    hasActivated(y, `Request-third-party-consent`(x, pat, id)),
    no-main-role-active(x),
    `PDS`$\Diamond$`PDS`.hasActivated(z, `Register-patient`(x))

(S2.2.11)
canDeactivate(x, x, `Third-party`()) ←

(S2.2.12)
isDeactivated(x, `Third-party`()) ←
    isDeactivated(y, `Request-third-party-consent`(x, pat, id)),
    other-third-party-consent-requests($0, y, x$)

(S2.2.13)
count-third-party-activations(count$\langle u \rangle$, $user$) ←
    hasActivated($user$, `Third-party`())

(S2.2.14)
canActivate(x, Third-party-consent(x, pat, id)) ←
    hasActivated(x, Third-party()),
    hasActivated(y, Request-third-party-consent(x, pat, id))

(S2.2.15)
canActivate(cli, Third-party-consent(x, pat, id)) ←
    hasActivated(cli, Spine-clinician($ra, org, spcty$)),
    canActivate(cli, Treating-clinician($pat, org, spcty$)),
    hasActivated(y, Request-third-party-consent(x, pat, id))

(S2.2.16)
isDeactivated(x, Third-party-consent(x, pat, id)) ←
    isDeactivated(y, Request-third-party-consent(x, pat, id)),
    other-third-party-consent-requests$(0, y, x)$

(S2.2.17)
third-party-consent(group$\langle consenter \rangle, pat, id$) ←
    hasActivated(x, Third-party-consent($consenter, pat, id$))

Consent to treatment

(S2.3.1)
canActivate($cli1$, Request-consent-to-treatment($pat, org2, cli2, spcty2$)) ←
    hasActivated($cli1$, Spine-clinician($ra1, org1, spcty1$)),
    canActivate($cli2$, Spine-clinician($ra2, org2, spcty2$)),
    canActivate(pat, Patient())

(S2.3.2)
canDeactivate($cli1, cli1$,
    Request-consent-to-treatment($pat, org2, cli2, spcty2$)) ←
    hasActivated($cli1$, Spine-clinician($ra1, org1, spcty1$))

(S2.3.3)
canDeactivate($cli2, cli1$,
    Request-consent-to-treatment($pat, org2, cli2, spcty2$)) ←
    hasActivated($cli2$, Spine-clinician($ra2, org2, spcty2$))

(S2.3.4)
canDeactivate(pat, x, Request-consent-to-treatment($pat, org, cli, spcty$)) ←
    hasActivated(pat, Patient())

(S2.3.5)
canDeactivate(ag, x, Request-consent-to-treatment($pat, org, cli, spcty$)) ←
    hasActivated(ag, Agent(pat))

(S2.3.6)
canDeactivate(cli, x, Request-consent-to-treatment($pat, org, cli2, spcty$)) ←
    hasActivated(cli, Spine-clinician($ra, org, spcty$)),
    canActivate(cli, General-practitioner(pat))

(S2.3.7)
isDeactivated(x, `Request-consent-to-treatment`($pat, org, cli, spcty$)) ←
    isDeactivated(y, `Register-patient`(pat))

(S2.3.8)
other-consent-to-treatment-requests(count$\langle y \rangle, x, pat, org, cli, spcty$) ←
    hasActivated(y, `Request-consent-to-treatment`($pat, org, cli, spcty$)),
    $x \neq y$

(S2.3.9)
canActivate(pat, `Consent-to-treatment`($pat, org, cli, spcty$)) ←
    hasActivated(pat, `Patient`()),
    hasActivated(x, `Request-consent-to-treatment`($pat, org, cli, spcty$))

(S2.3.10)
canActivate(ag, `Consent-to-treatment`($pat, org, cli, spcty$)) ←
    hasActivated(ag, `Agent`(pat)),
    hasActivated(x, `Request-consent-to-treatment`($pat, org, cli, spcty$))

(S2.3.11)
canActivate($cli1$, `Consent-to-treatment`($pat, org, cli2, spcty$)) ←
    hasActivated($cli1$, `Spine-clinician`($ra, org, spcty$)),
    canActivate($cli1$, `Treating-clinician`($pat, org, spcty$)),
    hasActivated(x, `Request-consent-to-treatment`($pat, org, cli2, spcty$))

(S2.3.12)
isDeactivated(x, `Consent-to-treatment`($pat, org, cli, spcty$)) ←
    isDeactivated(y, `Request-consent-to-treatment`($pat, org, cli, spcty$)),
    other-consent-to-treatment-requests$(0, y, pat, org, cli, spcty)$

Consent to group treatment

(S2.4.1)
canActivate(cli, `Request-consent-to-group-treatment`($pat, org, group$)) ←
    hasActivated(cli, `Spine-clinician`($ra, org, spcty$)),
    canActivate(pat, `Patient`())

(S2.4.2)
canDeactivate(cli, cli, `Request-consent-to-group-treatment`($pat, org, group$)) ←
    hasActivated(cli, `Spine-clinician`($ra, org, spcty$))

(S2.4.3)
canDeactivate(pat, x, `Request-consent-to-group-treatment`($pat, org, group$)) ←
    hasActivated(pat, `Patient`())

(S2.4.4)
canDeactivate(ag, x, `Request-consent-to-group-treatment`($pat, org, group$)) ←
    hasActivated(ag, `Agent`(pat))

(S2.4.5)
canDeactivate(cli, x, Request-consent-to-group-treatment($pat, org, group$)) ←
hasActivated(cli, Spine-clinician($ra, org, spcty$)),
canActivate(cli, General-practitioner(pat))

(S2.4.6)
canDeactivate(cli, x, Request-consent-to-group-treatment($pat, org, group$)) ←
hasActivated(cli, Spine-clinician($ra, org, spcty$)),
$ra \Diamond ra$.canActivate(cli, Workgroup-member($org, group, spcty$))

(S2.4.7)
isDeactivated(x, Request-consent-to-group-treatment($pat, org, group$)) ←
isDeactivated(y, Register-patient(pat))

(S2.4.8)
other-consent-to-group-treatment-requests(count$\langle y \rangle, x, pat, org, cli, spcty$) ←
hasActivated(y, Request-consent-to-group-treatment($pat, org, group$)),
$x \neq y$

(S2.4.9)
canActivate(pat, Consent-to-group-treatment($pat, org, group$)) ←
hasActivated(pat, Patient()),
hasActivated(x, Request-consent-to-group-treatment($pat, org, group$))

(S2.4.10)
canActivate(ag, Consent-to-group-treatment($pat, org, group$)) ←
hasActivated(ag, Agent(pat)),
hasActivated(x, Request-consent-to-group-treatment($pat, org, group$))

(S2.4.11)
canActivate($cli1$, Consent-to-group-treatment($pat, org, group$)) ←
hasActivated($cli1$, Spine-clinician($ra, org, spcty$)),
canActivate($cli1$, Treating-clinician($pat, org, spcty$)),
hasActivated(x, Request-consent-to-group-treatment($pat, org, group$))

(S2.4.12)
isDeactivated(x, Consent-to-group-treatment($pat, org, group$)) ←
isDeactivated(y, Request-consent-to-group-treatment($pat, org, group$)),
other-consent-to-group-treatment-requests$(0, y, pat, org, group)$

A.1.3 Legitimate Relationship

Referral

(S3.1.1)
canActivate($cli1$, Referrer($pat, org, cli2, spcty1$)) ←
hasActivated($cli1$, Spine-clinician($ra, org, spcty2$)),
canActivate($cli1$, Treating-clinician($pat, org, spcty2$))

(S3.1.2)
canDeactivate($cli1, cli1$, Referrer($pat, org, cli2, spcty1$)) ←

(S3.1.3)
canDeactivate($pat, cli1$, `Referrer`$(pat, org, cli2, spcty1)) \leftarrow$

(S3.1.4)
isDeactivated($cli1$, `Referrer`$(pat, org, cli2, spcty1)) \leftarrow$
isDeactivated(x, `Register-patient`(pat))

Emergency clinician

(S3.2.1)
canActivate(cli, `Spine-emergency-clinician`$(org, pat)) \leftarrow$
hasActivated(cli, `Spine-clinician`$(ra, org, spcty))$,
canActivate(pat, `Patient`$())$

(S3.2.2)
canDeactivate(cli, cli, `Spine-emergency-clinician`$(org, pat)) \leftarrow$

(S3.2.3)
isDeactivated(x, `Spine-emergency-clinician`$(org, pat)) \leftarrow$
isDeactivated(x, `Spine-clinician`$(ra, org, spcty)$)

(S3.2.4)
isDeactivated(x, `Spine-emergency-clinician`$(org, pat)) \leftarrow$
isDeactivated(y, `Register-patient`(pat))

Treating Clinician & GP

(S3.3.1)
canActivate(cli, `Treating-clinician`$(pat, org, spcty)) \leftarrow$
hasActivated(x, `Consent-to-treatment`$(pat, org, cli, spcty)$)

(S3.3.2)
canActivate(cli, `Treating-clinician`$(pat, org, spcty)) \leftarrow$
hasActivated(cli, `Spine-emergency-clinician`$(org, pat))$,
$spcty =$ `A-and-E`

(S3.3.3)
canActivate(cli, `Treating-clinician`$(pat, org, spcty)) \leftarrow$
canActivate(cli, `Spine-clinician`$(ra, org, spcty))$,
hasActivated(x, `Referrer`$(pat, org, cli, spcty)$)

(S3.3.4)
canActivate(cli, `Treating-clinician`$(pat, org, spcty)) \leftarrow$
canActivate(cli, `Group-treating-clinician`$(pat, ra, org, group, spcty)$)

(S3.3.5)
canActivate(cli, `General-practitioner`$(pat)) \leftarrow$
canActivate(cli, `Treating-clinician`$(pat, org, spcty))$,
$spcty =$ `GP`

Workgroup-based LR

(S3.4.1)
canActivate(cli, Group-treating-clinician($pat, ra, org, group, spcty$)) ←
    hasActivated(x, Consent-to-group-treatment($pat, org, group$)),
    ra.canActivate(cli, Workgroup-member($org, group, spcty$)),
    canActivate(ra, Registration-authority())

(S3.4.2)
canActivate(cli, Group-treating-clinician($pat, ra, org, group, spcty$)) ←
    hasActivated(x, Consent-to-group-treatment($pat, org, group$)),
    $ra \diamond ra$.canActivate(cli, Workgroup-member($org, group, spcty$)),
    canActivate(ra, Registration-authority())

A.1.4 Sealing-off data

Access restriction by clinician

(S4.1.1)
canActivate(cli, Concealed-by-spine-clinician($pat, ids, start, end$)) ←
    hasActivated(cli, Spine-clinician($ra, org, spcty$)),
    canActivate(cli, Treating-clinician($pat, org, spcty$))

(S4.1.2)
canDeactivate(cli, cli, Concealed-by-spine-clinician($pat, ids, start, end$)) ←
    hasActivated(cli, Spine-clinician($ra, org, spcty$))

(S4.1.3)
canDeactivate($cli, cli2$, Concealed-by-spine-clinician($pat, ids, start, end$)) ←
    hasActivated(cli, Spine-clinician($ra, org, spcty$)),
    canActivate(cli, General-practitioner(pat))

(S4.1.4)
canDeactivate($cli1, cli2$, Concealed-by-spine-clinician($pat, ids, start, end$)) ←
    hasActivated($cli1$, Spine-clinician($ra, org, spcty1$)),
    canActivate($cli1$, Group-treating-clinician($pat, ra, org, group, spcty1$)),
    canActivate($cli2$, Group-treating-clinician($pat, ra, org, group, spcty2$)),
    hasActivated(x, Consent-to-group-treatment($pat, org, group$))

(S4.1.5)
isDeactivated(x, Concealed-by-spine-clinician($pat, ids, start, end$)) ←
    isDeactivated(y, Register-patient(pat))

(S4.1.6)
count-concealed-by-spine-clinician(count$\langle x \rangle, pat, id$) ←
    hasActivated(x, Concealed-by-spine-clinician($pat, ids, start, end$)),
    $id \in ids$,
    Current-time() $\in [start, end]$

Access restriction by patient

(S4.2.1)
$\mathsf{canActivate}(pat, \texttt{Conceal-request}(what, who, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(pat, \texttt{Patient}()),$
$\quad \mathsf{count\text{-}conceal\text{-}requests}(n, pat),$
$\quad what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time),$
$\quad who = (orgs1, readers1, spctys1),$
$\quad n < 100$

(S4.2.2)
$\mathsf{canActivate}(ag, \texttt{Conceal-request}(what, who, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(ag, \texttt{Agent}(pat)),$
$\quad \mathsf{count\text{-}conceal\text{-}requests}(n, pat),$
$\quad what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time),$
$\quad who = (orgs1, readers1, spctys1),$
$\quad n < 100$

(S4.2.3)
$\mathsf{canDeactivate}(pat, x, \texttt{Conceal-request}(what, whom, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(pat, \texttt{Patient}()),$
$\quad \pi_1^7(what) = pat$

(S4.2.4)
$\mathsf{canDeactivate}(ag, x, \texttt{Conceal-request}(what, whom, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(ag, \texttt{Agent}(pat)),$
$\quad \pi_1^7(what) = pat$

(S4.2.5)
$\mathsf{canDeactivate}(cli, x, \texttt{Conceal-request}(what, whom, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{General-practitioner}(pat)),$
$\quad \pi_1^7(what) = pat$

(S4.2.6)
$\mathsf{isDeactivated}(x, \texttt{Conceal-request}(what, whom, start, end)) \leftarrow$
$\quad \mathsf{isDeactivated}(y, \texttt{Register-patient}(pat)),$
$\quad \pi_1^7(what) = pat$

(S4.2.7)
$\mathsf{count\text{-}conceal\text{-}requests}(\mathsf{count}\langle y \rangle, pat) \leftarrow$
$\quad \mathsf{hasActivated}(x, \texttt{Conceal-request}(y)),$
$\quad what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time),$
$\quad who = (orgs1, readers1, spctys1),$
$\quad y = (what, who, start, end)$

(S4.2.8)
$\mathsf{canActivate}(cli, \texttt{Concealed-by-spine-patient}(what, who, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{Treating-clinician}(pat, org, spcty)),$
$\quad \mathsf{hasActivated}(x, \texttt{Conceal-request}(what, who, start, end))$

(S4.2.9)
canDeactivate($cli, cli,$ `Concealed-by-spine-patient`$(what, who, start, end)) \leftarrow$
  hasActivated($cli,$ `Spine-clinician`$(ra, org, spcty)$)

(S4.2.10)
canDeactivate($cli1, cli2,$ `Concealed-by-spine-patient`$(what, who, start1, end1)) \leftarrow$
  hasActivated($cli1,$ `Spine-clinician`$(ra, org, spcty1)$),
  $ra \Diamond ra$.canActivate($cli1$,
    `Group-treating-clinician`$(pat, ra, org, group, spcty1)$),
  $ra \Diamond ra$.canActivate($cli2$,
    `Group-treating-clinician`$(pat, ra, org, group, spcty2)$)

(S4.2.11)
isDeactivated($cli,$ `Concealed-by-spine-patient`$(what, who, start, end)) \leftarrow$
  isDeactivated($x,$ `Conceal-request`$(what, who, start, end)$)

(S4.2.12)
count-concealed-by-spine-patient(count$\langle x \rangle, a, b) \leftarrow$
  hasActivated($x,$ `Concealed-by-spine-patient`$(what, who, start, end)$),
  $a = (pat, id)$,
  $b = (org, reader, spcty)$,
  $what = (pat, ids, orgs, authors, subjects, from\text{-}time, to\text{-}time)$,
  $whom = (orgs1, readers1, spctys1)$,
  `Get-spine-record-org`$(pat, id) \in orgs$,
  `Get-spine-record-author`$(pat, id) \in authors$,
  $sub \in$ `Get-spine-record-subjects`(pat, id),
  $sub \in subjects$,
  `Get-spine-record-time`$(pat, id) \in [from\text{-}time, to\text{-}time]$,
  $id \in ids$,
  $org \in orgs1$,
  $reader \in readers1$,
  $spcty \in spctys1$,
  `Current-time`$() \in [start, end]$,
  `Get-spine-record-third-parties`$(pat, id) = \{\}$,
  `non-clinical` $\in \Omega -$ `Get-spine-record-subjects`(pat, id)

Authenticated express consent

(S4.3.1)
canActivate($pat,$ `Authenticated-express-consent`$(pat, cli)) \leftarrow$
  hasActivated($pat,$ `Patient`$()$),
  count-authenticated-express-consent(n, pat),
  $n < 100$

(S4.3.2)
canActivate($ag,$ `Authenticated-express-consent`$(pat, cli)) \leftarrow$
  hasActivated($ag,$ `Agent`(pat)),
  count-authenticated-express-consent(n, pat),
  $n < 100$

(S4.3.3)
canActivate($cli1$, Authenticated-express-consent(pat, $cli2$)) ←
    hasActivated($cli1$, Spine-clinician(ra, org, $spcty$)),
    canActivate($cli1$, General-practitioner(pat))

(S4.3.4)
canDeactivate(pat, x, Authenticated-express-consent(pat, cli)) ←
    hasActivated(pat, Patient())

(S4.3.5)
canDeactivate(ag, x, Authenticated-express-consent(pat, cli)) ←
    hasActivated(ag, Agent(pat))

(S4.3.6)
canDeactivate($cli1$, x, Authenticated-express-consent(pat, $cli2$)) ←
    hasActivated($cli1$, Spine-clinician(ra, org, $spcty$)),
    canActivate($cli1$, General-practitioner(pat))

(S4.3.7)
isDeactivated(x, Authenticated-express-consent(pat, cli)) ←
    isDeactivated(y, Register-patient(pat))

(S4.3.8)
count-authenticated-express-consent(count⟨cli⟩, pat) ←
    hasActivated(x, Authenticated-express-consent(pat, cli))

A.1.5 Access permissions

Adding item

(S5.1.1)
permits(cli, Add-spine-record-item(pat)) ←
    hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
    canActivate(cli, Treating-clinician(pat, org, $spcty$))

(S5.1.2)
permits(pat, Annotate-spine-record-item(pat, id)) ←
    hasActivated(pat, Patient())

(S5.1.3)
permits(ag, Annotate-spine-record-item(pat, id)) ←
    hasActivated(ag, Agent(pat))

(S5.1.4)
permits(pat, Annotate-spine-record-item(pat, id)) ←
    hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
    canActivate(cli, Treating-clinician(pat, org, $spcty$))

Reading item IDs

(S5.2.1)
$\mathsf{permits}(pat, \texttt{Get-spine-record-item-ids}(pat)) \leftarrow$
$\quad \mathsf{hasActivated}(pat, \texttt{Patient}())$

(S5.2.2)
$\mathsf{permits}(ag, \texttt{Get-spine-record-item-ids}(pat)) \leftarrow$
$\quad \mathsf{hasActivated}(ag, \texttt{Agent}(pat))$

(S5.2.3)
$\mathsf{permits}(cli, \texttt{Get-spine-record-item-ids}(pat)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty)),$
$\quad \mathsf{canActivate}(cli, \texttt{Treating-clinician}(pat, org, spcty))$

Reading items

(S5.3.1)
$\mathsf{permits}(pat, \texttt{Read-spine-record-item}(pat, id)) \leftarrow$
$\quad \mathsf{hasActivated}(pat, \texttt{Patient}()),$
$\quad \mathsf{hasActivated}(x, \texttt{One-off-consent}(pat)),$
$\quad \mathsf{count\text{-}concealed\text{-}by\text{-}spine\text{-}patient}(n, a, b),$
$\quad \mathsf{count\text{-}concealed\text{-}by\text{-}spine\text{-}clinician}(m, pat, id),$
$\quad \mathsf{third\text{-}party\text{-}consent}(consenters, pat, id),$
$\quad n = 0,$
$\quad m = 0,$
$\quad a = (pat, id),$
$\quad b = (\texttt{No-org}, pat, \texttt{No-spcty}),$
$\quad \texttt{Get-spine-record-third-parties}(pat, id) \subseteq consenters$

(S5.3.2)
$\mathsf{permits}(ag, \texttt{Read-spine-record-item}(pat, id)) \leftarrow$
$\quad \mathsf{hasActivated}(ag, \texttt{Agent}(pat)),$
$\quad \mathsf{hasActivated}(x, \texttt{One-off-consent}(pat)),$
$\quad \mathsf{count\text{-}concealed\text{-}by\text{-}spine\text{-}patient}(n, a, b),$
$\quad \mathsf{count\text{-}concealed\text{-}by\text{-}spine\text{-}clinician}(m, pat, id),$
$\quad \mathsf{third\text{-}party\text{-}consent}(consenters, pat, id),$
$\quad n = 0,$
$\quad m = 0,$
$\quad a = (pat, id),$
$\quad b = (\texttt{No-org}, ag, \texttt{No-spcty}),$
$\quad \texttt{Get-spine-record-third-parties}(pat, id) \subseteq consenters$

(S5.3.3)
$\mathsf{permits}(cli, \texttt{Read-spine-record-item}(pat, id)) \leftarrow$
$\quad \mathsf{hasActivated}(cli, \texttt{Spine-clinician}(ra, org, spcty)),$
$\quad \mathsf{hasActivated}(x, \texttt{One-off-consent}(pat)),$
$\quad \texttt{Get-spine-record-org}(pat, id) = org,$
$\quad \texttt{Get-spine-record-author}(pat, id) = cli$

(S5.3.4)
permits(cli, Read-spine-record-item(pat, id)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
hasActivated(x, One-off-consent(pat)),
canActivate(cli, Treating-clinician(pat, org, $spcty$)),
count-concealed-by-spine-patient(n, a, b),
$n = 0$,
$a = (pat, id)$,
$b = (org, cli, spcty)$,
Get-spine-record-subjects(pat, id) ⊆ Permitted-subjects($spcty$)

(S5.3.5)
permits(cli, Read-spine-record-item(pat, id)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
hasActivated(x, One-off-consent(pat)),
canActivate(cli, Treating-clinician(pat, org, $spcty$)),
hasActivated(y, Authenticated-express-consent(pat, cli)),
Get-spine-record-subjects(pat, id) ⊆ Permitted-subjects($spcty$)

(S5.3.6)
permits(cli, Force-read-spine-record-item(pat, id)) ←
hasActivated(cli, Spine-clinician(ra, org, $spcty$)),
canActivate(cli, Treating-clinician(pat, org, $spcty$))

A.2 Policy for Patient Demographic Service

A.2.1 Main roles

Administrator

(P1.1.1)
canActivate(adm, PDS-manager()) ←
hasActivated(x, Register-PDS-manager(adm)),
no-main-role-active(adm)

(P1.1.2)
canDeactivate(adm, adm, PDS-manager()) ←

(P1.1.3)
isDeactivated(adm, PDS-manager()) ←
isDeactivated(x, Register-PDS-manager(adm))

(P1.1.4)
count-PDS-manager-activations(count⟨u⟩, $user$) ←
hasActivated($user$, PDS-manager())

(P1.1.5)
canActivate($adm1$, Register-PDS-manager($adm2$)) ←
hasActivated($adm1$, PDS-manager()),
pds-admin-regs(0, $adm2$)

(P1.1.6)
canDeactivate($adm1, x$, Register-PDS-manager($adm2$)) ←
    hasActivated($adm1$, PDS-manager())

(P1.1.7)
pds-admin-regs(count$\langle x \rangle, adm$) ←
    hasActivated(x, Register-PDS-manager(adm))

Patient

(P1.2.1)
canActivate(pat, Patient()) ←
    hasActivated(x, Register-patient(pat)),
    no-main-role-active(pat)

(P1.2.2)
canDeactivate(pat, pat, Patient()) ←

(P1.2.3)
isDeactivated(pat, Patient()) ←
    isDeactivated(x, Register-patient(pat))

(P1.2.4)
count-patient-activations(count$\langle u \rangle, user$) ←
    hasActivated($user$, Patient())

Agent

(P1.3.1)
canActivate(ag, Agent(pat)) ←
    hasActivated(x, Register-patient(ag)),
    no-main-role-active(ag),
    Spine$\Diamond$Spine.canActivate(ag, Agent(pat))

(P1.3.2)
canDeactivate(ag, ag, Agent(pat)) ←

(P1.3.3)
isDeactivated(ag, Agent(pat)) ←
    isDeactivated(x, Register-patient(ag))

(P1.3.4)
isDeactivated(ag, Agent(pat)) ←
    isDeactivated(x, Register-patient(pat))

(P1.3.5)
count-agent-activations(count$\langle u \rangle, user$) ←
    hasActivated($user$, Agent(pat))

NHS staff

(P1.4.1)
canActivate(x, `Professional-user`(ra, org)) ←
no-main-role-active(cli),
ra.hasActivated(x, `NHS-clinician-cert`($org, cli, spcty, start, end$)),
canActivate(ra, `Registration-authority`()),
`Current-time`() ∈ [$start, end$]

(P1.4.2)
canActivate(x, `Professional-user`(ra, org)) ←
no-main-role-active(cli),
$ra \Diamond ra$.hasActivated(x, `NHS-clinician-cert`($org, cli, spcty, start, end$)),
canActivate(ra, `Registration-authority`()),
`Current-time`() ∈ [$start, end$]

(P1.4.3)
canActivate(x, `Professional-user`(ra, org)) ←
no-main-role-active(cg),
ra.hasActivated(x, `NHS-Caldicott-guardian-cert`($org, cg, start, end$)),
canActivate(ra, `Registration-authority`()),
`Current-time`() ∈ [$start, end$]

(P1.4.4)
canActivate(x, `Professional-user`(ra, org)) ←
no-main-role-active(cg),
$ra \Diamond ra$.hasActivated(x, `NHS-Caldicott-guardian-cert`($org, cg, start, end$)),
canActivate(ra, `Registration-authority`()),
`Current-time`() ∈ [$start, end$]

(P1.4.5)
canDeactivate(x, x, `Professional-user`(ra, org)) ←

(P1.4.6)
count-professional-user-activations(count$\langle u \rangle$, $user$) ←
hasActivated($user$, `Professional-user`(ra, org))

Other

(P1.5.1)
no-main-role-active($user$) ←
count-agent-activations($n, user$),
count-patient-activations($n, user$),
count-PDS-manager-activations($n, user$),
count-preofessional-user-activations($n, user$),
$n = 0$

(P1.5.2)
canActivate(ra, `Registration-authority`()) ←
`NHS`.hasActivated(x, `NHS-registration-authority`($ra, start, end$)),
`Current-time`() ∈ [$start, end$]

(P1.5.3)
canActivate(ra, Registration-authority()) ←
ra◇NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end)),
Current-time() ∈ [$start$, end]

A.2.2 Patient registration

Registration

(P2.1.1)
canActivate(adm, Register-patient(pat)) ←
hasActivated(adm, PDS-manager()),
patient-regs(0, pat)

(P2.1.2)
canDeactivate(adm, x, Register-patient(pat)) ←
hasActivated(adm, PDS-manager())

(P2.1.3)
patient-regs(count⟨x⟩, pat) ←
hasActivated(x, Register-patient(pat))

Credentials

(P2.2.1)
canReqCred(pat, PDS.hasActivated(x, Register-patient(pat))) ←
hasActivated(pat, Patient())

(P2.2.2)
canReqCred(ag, PDS.hasActivated(x, Register-patient(pat))) ←
hasActivated(ag, Agent(pat))

(P2.2.3)
canReqCred(usr, PDS.hasActivated(x, Register-patient(pat))) ←
hasActivated(usr, Professional-user(ra, org))

(P2.2.4)
canReqCred(org, PDS.hasActivated(x, Register-patient(pat))) ←
ra.hasActivated(x, NHS-health-org-cert(org, $start$, end)),
canActivate(ra, Registration-authority())

(P2.2.5)
canReqCred(org, PDS.hasActivated(x, Register-patient(pat))) ←
org◇ra.hasActivated(x, NHS-health-org-cert(org, $start$, end)),
canActivate(ra, Registration-authority())

(P2.2.6)
canReqCred(ra, PDS.hasActivated(x, Register-patient(pat))) ←
canActivate(ra, Registration-authority())

(P2.2.7)
canReqCred(Spine, PDS.hasActivated(x, Register-patient(pat))) ←

A.3 Policy for Addenbrooke's Hospital

A.3.1 Main access roles

Clinician

(A1.1.1)
canActivate(mgr, Register-clinician($cli, spcty$)) ←
    hasActivated(mgr, HR-mgr()),
    clinician-regs($0, cli, spcty$)

(A1.1.2)
canDeactivate(mgr, x, Register-clinician($cli, spcty$)) ←
    hasActivated(mgr, HR-mgr())

(A1.1.3)
clinician-regs(count$\langle x \rangle, cli, spcty$) ←
    hasActivated(x, Register-clinician($cli, spcty$))

(A1.1.4)
canActivate(cli, Clinician($spcty$)) ←
    hasActivated(x, Register-clinician($cli, spcty$)),
    no-main-role-active(cli)

(A1.1.5)
canDeactivate(cli, cli, Clinician($spcty$)) ←

(A1.1.6)
isDeactivated(cli, Clinician($spcty$)) ←
    isDeactivated(x, Register-clinician($cli, spcty$))

(A1.1.7)
count-clinician-activations(count$\langle u \rangle, user$) ←
    hasActivated($user$, Clinician($spcty$))

Caldicott Guardian

(A1.2.1)
canActivate(mgr, Register-Caldicott-guardian(cg)) ←
    hasActivated(mgr, HR-mgr()),
    cg-regs($0, cg$)

(A1.2.2)
canDeactivate(mgr, x, Register-Caldicott-guardian(cg)) ←
    hasActivated(mgr, HR-mgr())

(A1.2.3)
cg-regs(count$\langle x \rangle, cg$) ←
    hasActivated(x, Register-Caldicott-guardian(cg))

(A1.2.4)
canActivate(cg, `Caldicott-guardian`()) ←
    hasActivated(x, `Register-Caldicott-guardian`(cg)),
    no-main-role-active(cg)

(A1.2.5)
canDeactivate(cg, cg, `Caldicott-guardian`()) ←

(A1.2.6)
isDeactivated(cg, `Caldicott-guardian`()) ←
    isDeactivated(x, `Register-Caldicott-guardian`(cg))

(A1.2.7)
count-caldicott-guardian-activations(count$\langle u \rangle$, $user$) ←
    hasActivated($user$, `Caldicott-guardian`())

HR manager

(A1.3.1)
canActivate(mgr, `Register-HR-mgr`($mgr2$)) ←
    hasActivated(mgr, `HR-mgr`()),
    hr-manager-regs(0, mgr)

(A1.3.2)
canDeactivate(mgr, x, `Register-HR-mgr`($mgr2$)) ←
    hasActivated(mgr, `HR-mgr`())

(A1.3.3)
hr-manager-regs(count$\langle x \rangle$, mgr) ←
    hasActivated(x, `Register-HR-mgr`(mgr))

(A1.3.4)
canActivate(mgr, `HR-mgr`()) ←
    hasActivated(x, `Register-HR-mgr`(mgr)),
    no-main-role-active(mgr)

(A1.3.5)
canDeactivate(mgr, mgr, `HR-mgr`()) ←

(A1.3.6)
isDeactivated(mgr, `HR-mgr`()) ←
    isDeactivated(x, `Register-HR-mgr`(mgr))

(A1.3.7)
count-hr-mgr-activations(count$\langle u \rangle$, $user$) ←
    hasActivated($user$, `HR-mgr`())

Receptionist

(A1.4.1)
canActivate(mgr, `Register-receptionist`(rec)) ←
    hasActivated(mgr, `HR-mgr`()),
    receptionist-regs($0, rec$)

(A1.4.2)
canDeactivate(mgr, x, `Register-receptionist`(rec)) ←
    hasActivated(mgr, `HR-mgr`())

(A1.4.3)
receptionist-regs(count$\langle x \rangle, rec$) ←
    hasActivated(x, `Register-receptionist`(rec))

(A1.4.4)
canActivate(rec, `Receptionist`()) ←
    hasActivated(x, `Register-receptionist`(rec))

(A1.4.5)
canDeactivate(rec, rec, `Receptionist`()) ←

(A1.4.6)
isDeactivated(rec, `Receptionist`()) ←
    isDeactivated(x, `Register-receptionist`(rec)),
    no-main-role-active(rec)

(A1.4.7)
count-receptionist-activations(count$\langle u \rangle, user$) ←
    hasActivated($user$, `Receptionist`())

Patient

(A1.5.1)
canActivate(rec, `Register-patient`(pat)) ←
    hasActivated(rec, `Receptionist`()),
    patient-regs($0, pat$)

(A1.5.2)
canDeactivate(rec, x, `Register-patient`(pat)) ←
    hasActivated(rec, `Receptionist`())

(A1.5.3)
patient-regs(count$\langle x \rangle, pat$) ←
    hasActivated(x, `Register-patient`(pat))

(A1.5.4)
canActivate(pat, `Patient`()) ←
    hasActivated(x, `Register-patient`(pat)),
    no-main-role-active(pat),
    `PDS`$\Diamond$`PDS`.hasActivated(y, `Register-patient`(pat))

(A1.5.5)
canDeactivate(pat, pat, Patient()) ←

(A1.5.6)
isDeactivated(pat, Patient()) ←
    isDeactivated(x, Register-patient(pat))

(A1.5.7)
count-patient-activations(count⟨u⟩, $user$) ←
    hasActivated($user$, Patient())

Agent

(A1.6.1)
canActivate($agent$, Agent(pat)) ←
    hasActivated(x, Register-agent($agent$, pat)),
    PDS◊PDS.hasActivated(x, Register-patient($agent$)),
    no-main-role-active($agent$)

(A1.6.2)
canActivate($agent$, Agent(pat)) ←
    canActivate(pat, Patient()),
    no-main-role-active($agent$),
    PDS◊PDS.hasActivated(x, Register-patient($agent$)),
    Spine◊Spine.canActivate($agent$, Agent(pat))

(A1.6.3)
isDeactivated(ag, Agent(pat)) ←
    isDeactivated(x, Register-agent(ag, pat)),
    other-agent-regs(0, x, ag, pat)

(A1.6.4)
count-agent-activations(count⟨u⟩, $user$) ←
    hasActivated($user$, Agent(pat))

(A1.6.5)
canActivate(pat, Register-agent($agent$, pat)) ←
    hasActivated(pat, Patient())

(A1.6.6)
canActivate(cg, Register-agent($agent$, pat)) ←
    hasActivated(cg, Caldicott-guardian()),
    canActivate(pat, Patient())

(A1.6.7)
canDeactivate(pat, pat, Register-agent($agent$, pat)) ←
    hasActivated(pat, Patient())

(A1.6.8)
canDeactivate(cg, x, Register-agent($agent$, pat)) ←
    hasActivated(cg, Caldicott-guardian())

(A1.6.9)
isDeactivated(x, Register-agent($agent, pat$)) ←
    isDeactivated(y, Register-patient(pat))

(A1.6.10)
other-agent-regs(count⟨y⟩, x, ag, pat) ←
    hasActivated(y, Register-agent(ag, pat)),
    $x \neq y$

Other

(A1.7.1)
no-main-role-active($user$) ←
    count-agent-activations($n, user$),
    count-caldicott-guardian-activations($n, user$),
    count-clinician-activations($n, user$),
    count-ext-treating-clinician-activations($n, user$),
    count-hr-mgr-activations($n, user$),
    count-patient-activations($n, user$),
    count-receptionist-activations($n, user$),
    count-third-party-activations($n, user$),
    $n = 0$

(A1.7.2)
canActivate(ra, Registration-authority()) ←
    NHS.hasActivated(x, NHS-registration-authority($ra, start, end$)),
    Current-time() ∈ [$start, end$]

(A1.7.3)
canActivate(ra, Registration-authority()) ←
    ra◇NHS.hasActivated(x, NHS-registration-authority($ra, start, end$)),
    Current-time() ∈ [$start, end$]

(A1.7.4)
canReqCred(x, RA-ADB.hasActivated(y, NHS-health-org-cert($org, start, end$))) ←
    org = ADB

A.3.2 Consent and referrals

Consent to referral

(A2.1.1)
canActivate($cli1$, Request-consent-to-referral($pat, ra, org, cli2, spcty2$)) ←
    hasActivated($cli1$, Clinician($spcty1$)),
    canActivate($cli1$, ADB-treating-clinician($pat, team, spcty1$))

(A2.1.2)
canDeactivate(cli, cli, Request-consent-to-referral($pat, ra, org, cli, spcty$)) ←
    hasActivated(cli, Clinician($spcty$))

(A2.1.3)
canDeactivate(pat, x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(pat, `Patient`())

(A2.1.4)
canDeactivate(ag, x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(ag, `Agent`(pat))

(A2.1.5)
canDeactivate(cg, x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(cg, `Caldicott-guardian`())

(A2.1.6)
isDeactivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    isDeactivated(y, `Register-patient`(pat))

(A2.1.7)
other-consent-to-referral-requests(count$\langle y \rangle, x, pat, ra, org, cli, spcty$) ←
    hasActivated(y, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)),
    $x \neq y$

(A2.1.8)
canActivate(pat, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(pat, `Patient`()),
    hasActivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$))

(A2.1.9)
canActivate(pat, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(pat, `Agent`(pat)),
    hasActivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$))

(A2.1.10)
canActivate(cg, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    hasActivated(cg, `Caldicott-guardian`()),
    hasActivated(x, `Request-consent-to-referral`($pat, ra, org, cli, spcty$))

(A2.1.11)
isDeactivated(x, `Consent-to-referral`($pat, ra, org, cli, spcty$)) ←
    isDeactivated(y, `Request-consent-to-referral`($pat, ra, org, cli, spcty$)),
    other-consent-to-referral-requests$(0, y, pat, ra, org, cli, spcty)$

(A2.1.12)
other-referral-consents(count$\langle y \rangle, x, pat, ra, org, cli, spcty$) ←
    hasActivated(y, `Consent-to-referral`($pat, ra, org, cli, spcty$)),
    $x \neq y$

External clinician

(A2.2.1)
canActivate(cli, Ext-treating-clinician($pat, ra, org, spcty$)) ←
    hasActivated(x, Consent-to-referral($pat, ra, org, cli, spcty$)),
    no-main-role-active(cli),
    ra.hasActivated(y, NHS-clinician-cert($org, cli, spcty, start, end$)),
    canActivate(ra, Registration-authority())

(A2.2.2)
canActivate(cli, Ext-treating-clinician($pat, ra, org, spcty$)) ←
    hasActivated(ref, Consent-to-referral($pat, ra, org, cli, spcty$)),
    no-main-role-active(cli),
    $ra \diamondsuit ra$.hasActivated(y, NHS-clinician-cert($org, cli, spcty, start, end$)),
    canActivate(ra, Registration-authority())

(A2.2.3)
canDeactivate(cli, cli, Ext-treating-clinician($pat, ra, org, spcty$)) ←

(A2.2.4)
isDeactivated(cli, Ext-treating-clinician($pat, ra, org, spcty$)) ←
    isDeactivated(x, Consent-to-referral($pat, ra, org, cli2, spcty$)),
    other-referral-consents($0, x, pat, ra, org, cli, spcty$)

(A2.2.5)
count-ext-treating-clinician-activations(count$\langle u \rangle$, $user$) ←
    hasActivated($user$, Ext-treating-clinician($pat, ra, org, spcty$))

Third-party consent

(A2.3.1)
canActivate(pat, Request-third-party-consent(x, pat, id)) ←
    hasActivated(pat, Patient()),
    $x \in$ Get-record-third-parties(pat, id)

(A2.3.2)
canActivate(ag, Request-third-party-consent(x, pat, id)) ←
    hasActivated(ag, Agent(pat)),
    $x \in$ Get-record-third-parties(pat, id)

(A2.3.3)
canActivate(cli, Request-third-party-consent(x, pat, id)) ←
    hasActivated(cli, Clinician($spcty$)),
    $x \in$ Get-record-third-parties(pat, id)

(A2.3.4)
canActivate(cg, Request-third-party-consent(x, pat, id)) ←
    hasActivated(cg, Caldicott-guardian()),
    $x \in$ Get-record-third-parties(pat, id)

(A2.3.5)
canDeactivate(pat, pat, Request-third-party-consent(x, pat, id)) ←
    hasActivated(pat, Patient())

(A2.3.6)
canDeactivate(ag, ag, Request-third-party-consent(x, pat, id)) ←
    hasActivated(pat, Agent(pat))

(A2.3.7)
canDeactivate(cli, cli, Request-third-party-consent(x, pat, id)) ←
    hasActivated(cli, Clinician($spcty$))

(A2.3.8)
canDeactivate(cg, x, Request-third-party-consent(y, pat, id)) ←
    hasActivated(cg, Caldicott-guardian())

(A2.3.9)
canDeactivate(x, y, Request-third-party-consent(x, pat, id)) ←
    hasActivated(x, Third-party())

(A2.3.10)
isDeactivated(x, Request-third-party-consent($x2$, pat, id)) ←
    isDeactivated(y, Register-patient(pat))

(A2.3.11)
count-third-party-activations(count$\langle u \rangle$, $user$) ←
    hasActivated($user$, Third-party())

(A2.3.12)
canActivate(x, Third-party()) ←
    hasActivated(y, Request-third-party-consent(x, pat, id)),
    no-main-role-active(x),
    PDS$\Diamond$PDS.hasActivated(z, Register-patient(x))

(A2.3.13)
canDeactivate(x, x, Third-party()) ←

(A2.3.14)
other-third-party-requests(count$\langle y \rangle$, x, $third\text{-}party$) ←
    hasActivated(y, Request-third-party-consent($third\text{-}party$, pat, id)),
    $x \neq y$

(A2.3.15)
isDeactivated(x, Third-party()) ←
    isDeactivated(y, Request-third-party-consent(x, pat, id)),
    other-third-party-requests(0, y, x)

(A2.3.16)
canActivate(x, Third-party-consent(x, pat, id)) ←
    hasActivated(x, Third-party()),
    hasActivated(y, Request-third-party-consent(x, pat, id))

(A2.3.17)
canActivate(cg, Third-party-consent(x, pat, id)) ←
    hasActivated(cg, Caldicott-guardian()),
    hasActivated(y, Request-third-party-consent(x, pat, id))

(A2.3.18)
canDeactivate(x, x, Third-party-consent(x, pat, id)) ←
    hasActivated(x, Third-party())

(A2.3.19)
canDeactivate(cg, x, Third-party-consent(x, pat, id)) ←
    hasActivated(cg, Caldicott-guardian())

(A2.3.20)
isDeactivated(x, Third-party-consent(x, pat, id)) ←
    isDeactivated(y, Register-patient(pat))

(A2.3.21)
third-party-consent(group⟨$consenter$⟩, pat, id) ←
    hasActivated(x, Third-party-consent($consenter$, pat, id))

A.3.3 LR and clinical teams

Head of team

(A3.1.1)
canActivate(hd, Head-of-team($team$)) ←
    hasActivated(x, Register-head-of-team(hd, $team$))

(A3.1.2)
canDeactivate(hd, hd, Head-of-team($team$)) ←

(A3.1.3)
isDeactivated(hd, Head-of-team($team$)) ←
    isDeactivated(x, Register-head-of-team(hd, $team$))

(A3.1.4)
canActivate(mgr, Register-head-of-team(hd, $team$)) ←
    hasActivated(mgr, HR-mgr()),
    hasActivated(x, Register-team-member(hd, $team$, $spcty$)),
    head-of-team-regs(0, hd, $team$)

(A3.1.5)
canDeactivate(mgr, x, Register-head-of-team(hd, $team$)) ←
    hasActivated(mgr, HR-mgr())

(A3.1.6)
isDeactivated(x, Register-head-of-team(hd, $team$)) ←
    isDeactivated(y, Register-team-member(hd, $team$, $spcty$))

(A3.1.7)
head-of-team-regs(count⟨x⟩, hd, $team$) ←
    hasActivated(x, Register-head-of-team(hd, $team$))

Team membership

(A3.2.1)
canActivate(mgr, Register-team-member(mem, $team$, $spcty$)) ←
    hasActivated(mgr, HR-mgr()),
    canActivate(mem, Clinician($spcty$)),
    team-member-regs(0, mem, $team$, $spcty$)

(A3.2.2)
canActivate(hd, Register-team-member(mem, $team$, $spcty$)) ←
    hasActivated(hd, Clinician($spcty2$)),
    canActivate(hd, Head-of-team($team$)),
    canActivate(mem, Clinician($spcty$)),
    team-member-regs(0, mem, $team$, $spcty$)

(A3.2.3)
canDeactivate(mgr, x, Register-team-member(mem, $team$, $spcty$)) ←
    hasActivated(mgr, HR-mgr())

(A3.2.4)
canDeactivate(hd, x, Register-team-member(mem, $team$, $spcty$)) ←
    hasActivated(hd, Clinician($spcty2$)),
    canActivate(hd, Head-of-team($team$))

(A3.2.5)
isDeactivated(x, Register-team-member(mem, $team$, $spcty$)) ←
    isDeactivated(y, Register-clinician(mem, $spcty$))

(A3.2.6)
canReqCred(ra, ADB.hasActivated(x, Register-team-member(cli, $team$, $spcty$))) ←
    ra = RA-ADB

(A3.2.7)
team-member-regs(count$\langle x \rangle$, mem, $team$, $spcty$) ←
    hasActivated(x, Register-team-member(mem, $team$, $spcty$))

Team episode

(A3.3.1)
canActivate(rec, Register-team-episode(pat, $team$)) ←
    hasActivated(rec, Receptionist()),
    canActivate(pat, Patient()),
    team-episode-regs(0, pat, $team$)

(A3.3.2)
canActivate(cli, Register-team-episode(pat, $team$)) ←
    hasActivated(cli, Clinician($spcty$)),
    hasActivated(x, Register-team-member(cli, $team$, $spcty$)),
    canActivate(pat, Patient()),
    team-episode-regs(0, pat, $team$)

(A3.3.3)
canDeactivate(cg, x, `Register-team-episode`($pat, team$)) ←
    hasActivated(cg, `Caldicott-guardian`())

(A3.3.4)
canDeactivate(rec, x, `Register-team-episode`($pat, team$)) ←
    hasActivated(rec, `Receptionist`())

(A3.3.5)
canDeactivate(cli, x, `Register-team-episode`($pat, team$)) ←
    hasActivated(cli, `Clinician`($spcty$)),
    hasActivated(x, `Register-team-member`($cli, team, spcty$))

(A3.3.6)
isDeactivated(x, `Register-team-episode`($pat, team$)) ←
    isDeactivated(y, `Register-patient`(pat))

(A3.3.7)
team-episode-regs(count$\langle x \rangle, pat, team$) ←
    hasActivated(x, `Register-team-episode`($pat, team$))

Head of ward

(A3.4.1)
canActivate(cli, `Head-of-ward`($ward$)) ←
    hasActivated(x, `Register-head-of-ward`($cli, ward$))

(A3.4.2)
canDeactivate(cli, cli, `Head-of-ward`($ward$)) ←

(A3.4.3)
isDeactivated(cli, `Head-of-ward`($ward$)) ←
    isDeactivated(x, `Register-head-of-ward`($cli, ward$))

(A3.4.4)
canActivate(mgr, `Register-head-of-ward`($cli, ward$)) ←
    hasActivated(mgr, `HR-mgr`()),
    hasActivated(x, `Register-ward-member`($cli, ward, spcty$)),
    head-of-ward-regs($0, cli, ward$)

(A3.4.5)
canDeactivate(mgr, x, `Register-head-of-ward`($cli, ward$)) ←
    hasActivated(mgr, `HR-mgr`())

(A3.4.6)
isDeactivated(x, `Register-head-of-ward`($cli, ward$)) ←
    isDeactivated(y, `Register-ward-member`($cli, ward, spcty$))

(A3.4.7)
head-of-ward-regs(count$\langle x \rangle, cli, ward$) ←
    hasActivated(x, `Register-head-of-ward`($cli, ward$))

Ward membership

(A3.5.1)
canActivate(mgr, Register-ward-member($cli, ward, spcty$)) ←
    hasActivated(mgr, HR-mgr()),
    canActivate(cli, Clinician($spcty$)),
    ward-member-regs($0, cli, ward, spcty$)

(A3.5.2)
canActivate(hd, Register-ward-member($cli, ward, spcty$)) ←
    hasActivated(cli, Clinician($spcty2$)),
    canActivate(hd, Head-of-ward($ward$)),
    canActivate(cli, Clinician($spcty$)),
    ward-member-regs($0, cli, ward, spcty$)

(A3.5.3)
canDeactivate(mgr, x, Register-ward-member($cli, ward, spcty$)) ←
    hasActivated(mgr, HR-mgr())

(A3.5.4)
canDeactivate(hd, x, Register-ward-member($cli, ward, spcty$)) ←
    hasActivated(hd, Clinician($spcty2$)),
    canActivate(hd, Head-of-ward($ward$))

(A3.5.5)
canReqCred(ra, ADB.hasActivated(x, Register-ward-member($cli, ward, spcty$))) ←
    ra = RA-ADB

(A3.5.6)
isDeactivated(x, Register-ward-member($cli, ward, spcty$)) ←
    isDeactivated(y, Register-clinician($cli, spcty$))

(A3.5.7)
ward-member-regs(count$\langle x \rangle, cli, ward, spcty$) ←
    hasActivated(x, Register-ward-member($cli, ward, spcty$))

Ward episode

(A3.6.1)
canActivate(rec, Register-ward-episode($pat, ward$)) ←
    hasActivated(rec, Receptionist()),
    canActivate(pat, Patient()),
    ward-episode-regs($0, pat, ward$)

(A3.6.2)
canActivate(hd, Register-ward-episode($pat, ward$)) ←
    hasActivated(hd, Clinician($spcty$)),
    canActivate(hd, Head-of-ward($ward$)),
    canActivate(pat, Patient()),
    ward-episode-regs($0, pat, ward$)

(A3.6.3)
canDeactivate(cg, x, `Register-ward-episode`($pat, ward$)) ←
    hasActivated(cg, `Caldicott-guardian`())

(A3.6.4)
canDeactivate(rec, x, `Register-ward-episode`($pat, ward$)) ←
    hasActivated(rec, `Receptionist`())

(A3.6.5)
canDeactivate(hd, x, `Register-ward-episode`($pat, ward$)) ←
    hasActivated(hd, `Clinician`($spcty$)),
    canActivate(hd, `Head-of-ward`($ward$))

(A3.6.6)
isDeactivated(x, `Register-ward-episode`($pat, ward$)) ←
    isDeactivated(y, `Register-patient`(pat))

(A3.6.7)
ward-episode-regs(count$\langle x \rangle, pat, ward$) ←
    hasActivated(x, `Register-ward-episode`($pat, ward$))

Emergency clinician

(A3.7.1)
canActivate(cli, `Emergency-clinician`(pat)) ←
    hasActivated(cli, `Clinician`($spcty$)),
    canActivate(pat, `Patient`())

(A3.7.2)
canDeactivate(cli, cli, `Emergency-clinician`(pat)) ←

(A3.7.3)
canDeactivate(cg, cli, `Emergency-clinician`(pat)) ←
    hasActivated(cg, `Caldicott-guardian`())

(A3.7.4)
isDeactivated(x, `Emergency-clinician`(pat)) ←
    isDeactivated(y, `Register-patient`(pat))

(A3.7.5)
isDeactivated(x, `Emergency-clinician`(pat)) ←
    isDeactivated(x, `Clinician`($spcty$))

(A3.7.6)
is-emergency-clinician(group$\langle x \rangle, pat$) ←
    hasActivated(x, `Emergency-clinician`(pat))

Treating clinician

(A3.8.1)
canActivate(cli, ADB-treating-clinician($pat, group, spcty$)) ←
    canActivate(cli, Clinician($spcty$)),
    hasActivated(x, Register-team-member($cli, team, spcty$)),
    hasActivated(y, Register-team-episode($pat, team$)),
    $group = team$

(A3.8.2)
canActivate(cli, ADB-treating-clinician($pat, group, spcty$)) ←
    canActivate(cli, Clinician($spcty$)),
    hasActivated(x, Register-ward-member($cli, ward, spcty$)),
    hasActivated(x, Register-ward-episode($pat, ward$)),
    $group = ward$

(A3.8.3)
canActivate(cli, ADB-treating-clinician($pat, group, spcty$)) ←
    hasActivated(cli, Emergency-clinician(pat)),
    $group$ = A-and-E,
    $spcty$ = A-and-E

A.3.4 Sealing-off data

Access restriction by clinician

(A4.1.1)
canActivate(cli, Concealed-by-clinician($pat, id, start, end$)) ←
    hasActivated(cli, Clinician($spcty$)),
    canActivate(cli, ADB-treating-clinician($pat, group, spcty$))

(A4.1.2)
canDeactivate(cli, cli, Concealed-by-clinician($pat, id, start, end$)) ←
    hasActivated(cli, Clinician($spcty$))

(A4.1.3)
canDeactivate($cli1, cli2$, Concealed-by-clinician($pat, id, start, end$)) ←
    hasActivated($cli1$, Clinician($spcty1$)),
    canActivate($cli1$, ADB-treating-clinician($pat, group, spcty1$)),
    canActivate($cli2$, ADB-treating-clinician($pat, group, spcty2$))

(A4.1.4)
canDeactivate(cg, cli, Concealed-by-clinician($pat, id, start, end$)) ←
    hasActivated(cg, Caldicott-guardian())

(A4.1.5)
isDeactivated(x, Concealed-by-clinician($pat, id, start, end$)) ←
    isDeactivated(y, Register-patient(pat))

(A4.1.6)
count-concealed-by-clinician(count$\langle x \rangle, pat, id$) $\leftarrow$
hasActivated(x, `Concealed-by-clinician`($pat, id, start, end$)),
`Current-time`() $\in [start, end]$

Access restriction by patient

(A4.2.1)
canActivate(pat, `Concealed-by-patient`($what, who, start, end$)) $\leftarrow$
hasActivated(pat, `Patient`()),
count-concealed-by-patient(n, pat),
$what = (pat, ids, authors, groups, subjects, from\text{-}time, to\text{-}time)$,
$who = (orgs1, readers1, groups1, spctys1)$,
$n < 100$

(A4.2.2)
canActivate(ag, `Concealed-by-patient`($what, who, start, end$)) $\leftarrow$
hasActivated(ag, `Agent`(pat)),
count-concealed-by-patient(n, pat),
$what = (pat, ids, authors, groups, subjects, from\text{-}time, to\text{-}time)$,
$who = (orgs1, readers1, groups1, spctys1)$,
$n < 100$

(A4.2.3)
canDeactivate(pat, x, `Concealed-by-patient`($what, whom, start, end$)) $\leftarrow$
hasActivated(pat, `Patient`()),
$\pi_1^7(what) = pat$

(A4.2.4)
canDeactivate(ag, x, `Concealed-by-patient`($what, whom, start, end$)) $\leftarrow$
hasActivated(ag, `Agent`(pat)),
$\pi_1^7(what) = pat$

(A4.2.5)
canDeactivate(cg, x, `Concealed-by-patient`($what, whom, start, end$)) $\leftarrow$
hasActivated(cg, `Caldicott-guardian`())

(A4.2.6)
isDeactivated(x, `Concealed-by-patient`($what, whom, start, end$)) $\leftarrow$
isDeactivated(y, `Register-patient`(pat)),
$\pi_1^7(what) = pat$

(A4.2.7)
count-concealed-by-patient(count$\langle y \rangle, pat$) $\leftarrow$
hasActivated(x, `Concealed-by-patient`(y)),
$what = (pat, ids, authors, groups, subjects, from\text{-}time, to\text{-}time)$,
$who = (orgs1, readers1, groups1, spctys1)$,
$y = (what, who, start, end)$

(A4.2.8)
count-concealed-by-patient2(count$\langle x \rangle, a, b$) $\leftarrow$
hasActivated(x, `Concealed-by-patient`($what, whom, start, end$)),
$a = (pat, id)$,
$b = (org, reader, group, spcty)$,
$what = (pat, ids, authors, groups, subjects, from\text{-}time, to\text{-}time)$,
$whom = (orgs1, readers1, groups1, spctys1)$,
`Get-record-author`(pat, id) $\in authors$,
`Get-record-group`(pat, id) $\in groups$,
$sub \in$ `Get-record-subjects`(pat, id),
$sub \in subjects$,
`Get-record-time`(pat, id) $\in [from\text{-}time, to\text{-}time]$,
$id \in ids$,
$org \in orgs1$,
$reader \in readers1$,
$group \in groups1$,
$spcty \in spctys1$,
`Current-time`() $\in [start, end]$

A.3.5 Access permissions

Adding item

(A5.1.1)
permits(cli, `Add-record-item`(pat)) $\leftarrow$
hasActivated(cli, `Clinician`($spcty$)),
canActivate(cli, `ADB-treating-clinician`($pat, group, spcty$))

(A5.1.2)
permits(cli, `Add-record-item`(pat)) $\leftarrow$
hasActivated(cli, `Ext-treating-clinician`($pat, ra, org, spcty$))

(A5.1.3)
permits(ag, `Annotate-record-item`(pat, id)) $\leftarrow$
hasActivated(ag, `Agent`(pat))

(A5.1.4)
permits(pat, `Annotate-record-item`(pat, id)) $\leftarrow$
hasActivated(pat, `Patient`())

(A5.1.5)
permits(pat, `Annotate-record-item`(pat, id)) $\leftarrow$
hasActivated(cli, `Clinician`($spcty$)),
canActivate(cli, `ADB-treating-clinician`($pat, group, spcty$))

Reading item IDs

(A5.2.1)
permits(pat, `Get-record-item-ids`(pat)) $\leftarrow$
hasActivated(pat, `Patient`())

(A5.2.2)
permits(ag, Get-record-item-ids(pat)) ←
    hasActivated(ag, Agent(pat))

(A5.2.3)
permits(cli, Get-record-item-ids(pat)) ←
    hasActivated(cli, Clinician($spcty$)),
    canActivate(cli, ADB-treating-clinician(pat, $group$, $spcty$))

Reading items

(A5.3.1)
permits(ag, Read-record-item(pat, id)) ←
    hasActivated(ag, Agent(pat)),
    count-concealed-by-patient2(n, a, b),
    count-concealed-by-clinician(m, pat, id),
    third-party-consent($consenters$, pat, id),
    $a = (pat, id)$,
    $b =$ (No-org, ag, No-group, No-spcty),
    $n = 0$,
    $m = 0$,
    Get-record-third-parties(pat, id) $\subseteq$ $consenters$

(A5.3.2)
permits(cli, Read-record-item(pat, id)) ←
    hasActivated(cli, Clinician($spcty$)),
    Get-record-author(pat, id) = cli

(A5.3.3)
permits(cli, Read-record-item(pat, id)) ←
    hasActivated(cli, Clinician($spcty$)),
    hasActivated(x, Register-team-member(cli, $team$, $spcty$)),
    Get-record-group(pat, id) = $team$

(A5.3.4)
permits(cli, Read-record-item(pat, id)) ←
    hasActivated(cli, Clinician($spcty$)),
    canActivate(cli, ADB-treating-clinician(pat, $group$, $spcty$)),
    count-concealed-by-patient2(n, a, b),
    $n = 0$,
    $a = (pat, id)$,
    $b =$ (ADB, cli, $group$, $spcty$),
    Get-record-subjects(pat, id) $\subseteq$ Permitted-subjects($spcty$)

(A5.3.5)
permits(cli, Read-record-item(pat, id)) ←
hasActivated(cli, Ext-treating-clinician(pat, ra, org, $spcty$)),
count-concealed-by-patient2(n, a, b),
$n = 0$,
$a = (pat, id)$,
$b = (org, cli,$ Ext-group$, spcty)$,
Get-record-subjects(pat, id) $\subseteq$ Permitted-subjects($spcty$)

(A5.3.6)
permits(pat, Read-record-item(pat, id)) ←
hasActivated(pat, Patient()),
count-concealed-by-patient2(n, a, b),
count-concealed-by-clinician(m, pat, id),
third-party-consent($consenters$, pat, id),
$n = 0$,
$m = 0$,
$a = (pat, id)$,
$b =$ (No-org, pat, No-group, No-spcty),
Get-record-third-parties(pat, id) $\subseteq$ $consenters$

(A5.3.7)
permits(cg, Force-read-record-item(pat, id)) ←
hasActivated(cg, Caldicott-guardian())

(A5.3.8)
permits(cli, Force-read-record-item(pat, id)) ←
hasActivated(cli, Clinician($spcty$)),
canActivate(cli, ADB-treating-clinician(pat, $group$, $spcty$))

A.4 Policy for Addenbrooke's Registration Authority

A.4.1 Main roles

Administrator

(R1.1.1)
canActivate(mgr, Register-RA-manager($mgr2$)) ←
hasActivated(mgr, RA-manager()),
ra-manager-regs(0, $mgr2$)

(R1.1.2)
canDeactivate(mgr, x, Register-RA-manager($mgr2$)) ←
hasActivated(mgr, RA-manager())

(R1.1.3)
ra-manager-regs(count$\langle x \rangle$, mgr) ←
hasActivated(x, Register-RA-manager(mgr))

(R1.1.4)
canActivate(mgr, RA-manager()) ←
    hasActivated(x, Register-RA-manager(mgr))

(R1.1.5)
canDeactivate(mgr, mgr, RA-manager()) ←

(R1.1.6)
isDeactivated(mgr, RA-manager()) ←
    isDeactivated(x, Register-RA-manager(mgr))

Other

(R1.2.1)
canReqCred(x, NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end))) ←
    ra = RA-ADB

(R1.2.2)
canActivate(srv, NHS-service()) ←
    canActivate(srv, Registration-authority())

(R1.2.3)
canActivate(srv, NHS-service()) ←
    srv = Spine

(R1.2.4)
canActivate(ra, Registration-authority()) ←
    NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end)),
    Current-time() ∈ [$start$, end]

(R1.2.5)
canActivate(ra, Registration-authority()) ←
    ra◇NHS.hasActivated(x, NHS-registration-authority(ra, $start$, end)),
    Current-time() ∈ [$start$, end]

A.4.2 NHS staff authentication

Clinician

(R2.1.1)
canActivate(mgr, NHS-clinician-cert(org, cli, $spcty$, $start$, end)) ←
    hasActivated(mgr, RA-manager()),
    hasActivated(y, NHS-health-org-cert(org, $start2$, $end2$)),
    $start \in [start2, end2]$,
    $end \in [start2, end2]$,
    $start < end$

(R2.1.2)
canDeactivate(mgr, x, NHS-clinician-cert(org, cli, $spcty$, $start$, end)) ←
    hasActivated(mgr, RA-manager())

(R2.1.3)
$\mathsf{isDeactivated}(mgr, \texttt{NHS-clinician-cert}(org, cli, spcty, start, end)) \leftarrow$
$\quad \mathsf{isDeactivated}(x, \texttt{NHS-health-org-cert}(org, start2, end2)),$
$\quad \mathsf{other\text{-}NHS\text{-}health\text{-}org\text{-}regs}(0, x, org, start2, end2),$
$\quad start \in [start2, end2],$
$\quad end \in [start2, end2],$
$\quad start < end$

(R2.1.4)
$\mathsf{canReqCred}(org, \texttt{RA-ADB}.\mathsf{hasActivated}(x,$
$\quad \texttt{NHS-clinician-cert}(org, cli, spcty, start, end))) \leftarrow$
$\quad \mathsf{hasActivated}(y, \texttt{NHS-health-org-cert}(org, start2, end2)),$
$\quad \texttt{Current-time}() \in [start2, end2]$

(R2.1.5)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x,$
$\quad \texttt{NHS-clinician-cert}(org, cli, spcty, start, end))) \leftarrow$
$\quad \mathsf{canActivate}(e, \texttt{NHS-service}())$

(R2.1.6)
$\mathsf{canReqCred}(cli, \texttt{RA-ADB}.\mathsf{hasActivated}(x,$
$\quad \texttt{NHS-clinician-cert}(org, cli, spcty, start, end))) \leftarrow$

Caldicott Guardian

(R2.2.1)
$\mathsf{canActivate}(mgr, \texttt{NHS-Caldicott-guardian-cert}(org, cg, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(mgr, \texttt{RA-manager}()),$
$\quad \mathsf{hasActivated}(x, \texttt{NHS-health-org-cert}(org, start2, end2)),$
$\quad start \in [start2, end2],$
$\quad end \in [start2, end2],$
$\quad start < end$

(R2.2.2)
$\mathsf{canDeactivate}(mgr, x, \texttt{NHS-Caldicott-guardian-cert}(org, cg, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(mgr, \texttt{RA-manager}())$

(R2.2.3)
$\mathsf{isDeactivated}(mgr, \texttt{NHS-Caldicott-guardian-cert}(org, cg, start, end)) \leftarrow$
$\quad \mathsf{isDeactivated}(x, \texttt{NHS-health-org-cert}(org, start2, end2)),$
$\quad \mathsf{other\text{-}NHS\text{-}health\text{-}org\text{-}regs}(0, x, org, start2, end2),$
$\quad start \in [start2, end2],$
$\quad end \in [start2, end2],$
$\quad start < end$

(R2.2.4)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x,$
$\quad \texttt{NHS-Caldicott-guardian-cert}(org, cg, start, end))) \leftarrow$
$\quad e = cg$

(R2.2.5)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x,$
$\quad \texttt{NHS-Caldicott-guardian-cert}(org, cg, start, end))) \leftarrow$
$\quad \mathsf{hasActivated}(y, \texttt{NHS-health-org-cert}(org, start2, end2)),$
$\quad e = org,$
$\quad \texttt{Current-time}() \in [start2, end2]$

(R2.2.6)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x,$
$\quad \texttt{NHS-Caldicott-guardian-cert}(org, cg, start, end))) \leftarrow$
$\quad \mathsf{canActivate}(e, \texttt{NHS-service}())$

Health organisation

(R2.3.1)
$\mathsf{canActivate}(mgr, \texttt{NHS-health-org-cert}(org, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(mgr, \texttt{RA-manager}())$

(R2.3.2)
$\mathsf{canDeactivate}(mgr, x, \texttt{NHS-health-org-cert}(org, start, end)) \leftarrow$
$\quad \mathsf{hasActivated}(mgr, \texttt{RA-manager}())$

(R2.3.3)
$\mathsf{other\text{-}NHS\text{-}health\text{-}org\text{-}regs}(\mathsf{count}\langle y \rangle, x, org, start, end) \leftarrow$
$\quad \mathsf{hasActivated}(y, \texttt{NHS-health-org-cert}(org, start2, end2)),$
$\quad start \in [start2, end2],$
$\quad end \in [start2, end2],$
$\quad start < end,$
$\quad x \neq y \vee start \neq start2 \vee end \neq end2$

(R2.3.4)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x, \texttt{NHS-health-org-cert}(org, start, end))) \leftarrow$
$\quad \mathsf{hasActivated}(y, \texttt{NHS-Caldicott-guardian-cert}(org, cg, start2, end2)),$
$\quad \texttt{Current-time}() \in [start2, end2],$
$\quad e = cg$

(R2.3.5)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x, \texttt{NHS-health-org-cert}(org, start, end))) \leftarrow$
$\quad \mathsf{hasActivated}(y, \texttt{NHS-clinician-cert}(org, cli, spcty, start2, end2)),$
$\quad \texttt{Current-time}() \in [start2, end2],$
$\quad e = cli$

(R2.3.6)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x, \texttt{NHS-health-org-cert}(org, start, end))) \leftarrow$
$\quad e = org$

(R2.3.7)
$\mathsf{canReqCred}(e, \texttt{RA-ADB}.\mathsf{hasActivated}(x, \texttt{NHS-health-org-cert}(org2, start, end))) \leftarrow$
$\quad ra.\mathsf{hasActivated}(y, \texttt{NHS-health-org-cert}(org, start2, end2)),$
$\quad \mathsf{canActivate}(ra, \texttt{Registration-authority}()),$
$\quad e = org$

(R2.3.8)
canReqCred(e, RA-ADB.hasActivated(x, NHS-health-org-cert($org2, start, end$))) ←
$org \Diamond ra$.hasActivated(y, NHS-health-org-cert($org, start2, end2$)),
canActivate(ra, Registration-authority()),
$e = org$

(R2.3.9)
canReqCred(e, RA-ADB.hasActivated(x, NHS-health-org-cert($org, start, end$))) ←
canActivate(e, NHS-service())

A.4.3 Workgroup management

(R3.1.1)
canActivate(cli, Workgroup-member($org, group, spcty$)) ←
hasActivated(x, NHS-health-org-cert($org, start, end$)),
$org \Diamond org$.hasActivated(x, Register-team-member($cli, group, spcty$)),
Current-time() $\in [start, end]$

(R3.1.2)
canActivate(cli, Workgroup-member($org, group, spcty$)) ←
hasActivated(x, NHS-health-org-cert($org, start, end$)),
$org \Diamond org$.hasActivated(x, Register-ward-member($cli, group, spcty$)),
Current-time() $\in [start, end]$

(R3.1.3)
canReqCred(Spine, RA-ADB.canActivate(cli,
Workgroup-member($org, group, spcty$))) ←

Bibliography

[Aba98] Martin Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–22, 1998.

[AG99] Martin Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Addison-Wesley, 1995.

[Aik94] Alexander Aiken. Set constraints: Results, applications, and future directions. In *PPCP '94: Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming*, pages 326–335, London, UK, 1994. Springer-Verlag.

[And96a] Ross Anderson. Patient confidentiality — at risk from NHS-wide networking. In B. Richards and H. de Glanville, editors, *Current perspectives in healthcare computing*, pages 687–692. Weybridge: BJHC Books, 1996.

[And96b] Ross Anderson. A security policy model for clinical information systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 30–42, 1996.

[And99] Ross Anderson. Information technology in medical practice: Safety and privacy lessons from the United Kingdom. *Medical Journal of Australia*, 170:181–185, 1999.

[ARH97] Alfarez Abdul-Rahman and Stephen Hailes. A distributed trust model. In *NSPW '97: Proceedings of the 1997 workshop on New security paradigms*, pages 48–60, New York, NY, USA, 1997. ACM Press.

[Arn03] Sarah Arnott. Confidentiality is top priority for patients (08/10/03). *Computing*, 2003. See `http://www.computing.co.uk/news/1144171`.

[Arn04] Sarah Arnott. Stability problems for NHS data spine (30/09/04). *Computing*, 2004.

[Arn05] Sarah Arnott. Key supplier cut from NHS IT plan (01/06/05). *Computing*, 2005.

[AS99] Gail-Joon Ahn and Ravi Sandhu. The RSL99 language for role-based separation of duty constraints. In *Proceedings of the fourth ACM workshop on Role-based access control*, pages 43–54. ACM Press, 1999.

[AS00] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security*, 3(4):207–226, 2000.

[ASW04] Sudhir Agarwal, Barbara Sprick, and Sandra Wortmann. Credential based access control for semantic web services. In *American Association for Artificial Intelligence Spring Symposium Series*, March 2004.

[BBC+97] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saibi, and B. Werner. The Coq proof assistant reference manual — version v6.1. Technical report, INRIA, August 1997.

[BBC03] BBC News. 'Dissident operation' uncovered (02/07/03). 2003. See `http://news.bbc.co.uk/1/low/northern_ireland/3038852.stm`.

[Bec05] Moritz Y. Becker. A formal security policy for an NHS electronic health record service. Technical Report UCAM-CL-TR-628, University of Cambridge, Computer Laboratory, April 2005.

[Bel04] Andras Belokosztolszki. *Role-based access control administration*. PhD thesis, Computer Laboratory, University of Cambridge, 2004.

[BFK99a] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.

[BFK99b] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. The KeyNote trust management system, RFC 2704, September 1999. See `http://www.ietf.org/rfc/rfc2704.txt`.

[BFK99c] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[BL75] David E. Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, The MITRE Corporation, July 1975.

[BLR03] Arosha K. Bandara, Emil Lupu, and Alessandra Russo. Using event calculus to formalise policy specification and analysis. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.

[BMB+00] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic support for distributed applications. *Computer*, 33(3):68–76, 2000.

[BMY02] Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.

[Bre05] Sean Brennan. *The NHS IT Project: The biggest computer programme in the world... ever!* Radcliffe Publishing, 2005.

[Bro00] Anthony Browne. Lives ruined as NHS leaks patients' notes (25/06/00). *The Observer*, 2000. See `http://observer.guardian.co.uk/uk_news/story/0,6903,336271,00.html`.

[BS93] Naomi Baker and Harald Sondergaard. Definiteness analysis for CLP(R). In *Australian Computer Science Conference*, pages 321–332, 1993.

[BS00] Ezedin Barka and Ravi Sandhu. Framework for role-based delegation models. In *Proceedings of the 16th Annual Computer Security Applications Conference*. IEEE Computer Society, 2000.

[BS04a] Moritz Y. Becker and Peter Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, 2004.

[BS04b] Moritz Y. Becker and Peter Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations Workshop*, pages 139–154, 2004.

[Car05] John Carvel. Patients can stay off NHS database (14/01/05). *The Guardian*, 2005.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, 1977.

[CEE+01] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[Cla05a] Lindsay Clark. London NHS IT scheme chief resigns (03/05/05). *Computer Weekly*, 2005.

[Cla05b] Lindsay Clark. NHS IT leadership changes hands - again (20/05/05). *Computer Weekly*, 2005.

[CNS03] Marco Carbone, Mogens Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *International Conference on Software Engineering and Formal Methods (SEFM'03)*, 2003.

[Col03a] Tony Collins. Doctors express alarm at plans to store patient data without consent (15/07/03). *Computer Weekly*, 2003. See `http://www.computerweekly.com/Article123355.htm`.

[Col03b] Tony Collins. Health service's £2.3bn programme loses its head (01/12/03). *Computer Weekly*, 2003.

[Col03c] Tony Collins. How the national programme came to be the health service's riskiest IT project (16/09/03). *Computer Weekly*, 2003. See `http://www.computerweekly.com/Article124870.htm`.

[Col04a] Tony Collins. Final cost of NHS IT programme could rise to more than £18.6bn (12/10/04). *Computer Weekly*, 2004.

[Col04b] Tony Collins. GPs vote to boycott patient record database (29/06/04). *Computer Weekly*, 2004. See `http://www.computerweekly.com/Article131577.htm`.

[Col04c] Tony Collins. NHS joint IT chief resigns after six months in the job (28/09/04). *Computer Weekly*, 2004.

[Cor02] Amanda Cornwall. Electronic health records: an international perspective. *Health Issues*, 73, 2002.

[Cou96] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, 1996.

[Cro03] Michael Cross. NHS spree revealed (12/06/03). *The Guardian*, 2003. See `http://www.guardian.co.uk/online/story/0,3605,975139,00.html`.

[CS95] Fang Chen and Ravi S. Sandhu. Constraints for role-based access control. In *RBAC '95: Proceedings of the first ACM Workshop on Role-based access control*, 1995.

[CS03] Tony Collins and Mike Simons. NHS plan branded a 'farce' (03/06/03). *Computer Weekly*, 2003. See `http://www.computerweekly.com/Article122277.htm`.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–195, 1987.

[CW93] Weidong Chen and David S. Warren. Query evaluation under the well-founded semantics. In *Proceedings of the twelfth ACM Symposium on Principles of Database Systems*, pages 168–179. ACM Press, 1993.

[CW96] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[Dam02] Nicodemos Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, 2002.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The Ponder policy specification language. In *Policy Workshop*, 2001.

[Dep01a] Department of Health, UK. Building the information core: Implementing the NHS plan. 2001.

[Dep01b] Department of Health, UK. Building the information core: Protecting and using confidential patient information. 2001.

[Dep02] Department of Health, UK. Legal and policy constraints on electronic records. 2002.

[DeT02] John DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[EFL+99] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. SPKI certificate theory, RFC 2693, September 1999. See `http://www.ietf.org/rfc/rfc2693.txt`.

[Ell99] Carl M. Ellison. SPKI requirements, RFC 2692, September 1999. See `http://www.ietf.org/rfc/rfc2692.txt`.

[FB97] David Ferraiolo and John Barkley. Specifying and managing role-based access control within a corporate intranet. In *Proceedings of the second ACM workshop on Role-based access control*, pages 77–82. ACM Press, 1997.

[FGM05] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. In *Proc. ESOP*, April 2005.

[FHLW02] Dieter Fensel, James A. Hendler, Henry Lieberman, and Wolfgang Wahlster. *Spinning The Semantic Web*. MIT Press, 2002.

[FJ02] Tim Finin and Anupam Joshi. Agents, trust, and information access on the semantic web. *ACM SIGMOD Record*, 31(4):30–35, 2002.

[FK92] David Ferraiolo and Richard Kuhn. Role-based access controls. In *Proceedings of the 15th NIST National Computer Security Conference*, pages 554–563, 1992.

[Fou03] Foundation for Information Policy Research. NHS confidentiality consultation – FIPR response. February 2003. See `http://www.cl.cam.ac.uk/users/rja14/fiprmedconf.html`.

[FSG+01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and Systems Security (TISSEC)*, 4(3):224–274, 2001.

[Gau03] Nick Gaunt. Confidentiality and consent: Use cases applicable to shared electronic health record. *S&W Devon ERDIP Project*, 2003.

[GDL95] Maurizio Gabbrielli, Giovanna M. Dore, and Giorgio Levi. Observable semantics for constraint logic programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.

[GGF98] Virgil Gligor, Serban L. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *19th IEEE Symposium on Security and Privacy*, pages 172–183, 1998.

[GI97] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, pages 153–159, 1997.

[GJ00a] Carl Gunter and Trevor Jim. Generalized certificate revocation. In *Symposium on Principles of Programming Languages*, pages 316–329, 2000.

[GJ00b] Carl Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software - Practice and Experience*, 30(15):1609–1640, 2000.

[GL91] Maurizio Gabbrielli and Giorgio Levi. Modeling answer constraints in constraint logic programs. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 238–252, Paris, France, 1991. The MIT Press.

[GM93] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[GS00] Tyrone Grandison and Morris Sloman. A survey of trust in Internet applications. *IEEE Communications Surveys and Tutorials*, 3(4), 2000.

[Haw03] Nigel Hawkes. Patient records go online (21/07/03). *The Times Online*, 2003. See `http://www.timesonline.co.uk/newspaper/0,,2710-751992,00.html`.

[HBM98] Richard Hayton, Jean Bacon, and Ken Moody. OASIS: Access control in an open distributed environment. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 3–14, 1998.

[Hil02] David Hilbert. Mathematical problems. Lecture delivered before the International Congress of Mathematicians at Paris in 1900. *Bulletin of the American Mathematical Society*, 8:437–479, 1902.

[HW03] Joseph Y. Halpern and Vicky Weissman. Using first-order logic to reason about policies. In *CSFW*, pages 187–201, 2003.

[ITU00] ITU-T (Telecommunication Standardization Sector, International Telecommunication Union). ITU-T recommendation X.509: The directory - public-key and attribute certificate frameworks, 2000.

[Jim01] Trevor Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JMMS98] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. The semantics of constraint logic programs. *Journal of Logic Programming*, 37(1-3):1–46, 1998.

[JS01] Trevor Jim and Dan Suciu. Dynamically distributed query evaluation. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems*, pages 28–39. ACM Press, 2001.

[JSS97] Sushil Jajodia, Pierangela Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1997.

[JSSS01] Sushil Jajodia, Pierangela Samarati, Maria Luisa Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[JT99] Catholijn M. Jonker and Jan Treur. Formal analysis of models for the dynamics of trust based on experiences. In *MAAMAW '99: Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 221–231, 1999.

[Kei05] Brian Keighley. Confidentiality — the final betrayal. *British Medical Journal Career Focus*, 330(259), 2005.

[Koz94] Dexter Kozen. Set constraints and logic programming. In *Constraints in Computational Logics*, pages 302–303, 1994.

[Kuh97] D. Richard Kuhn. Mutual exclusion of roles as a means of implementing separation of duty in role-based access control systems. In *Proceedings of the second ACM workshop on Role-based access control*, pages 23–30. ACM Press, 1997.

[Lam71] Butler Lampson. Protection. In *Proceedings of the Fifth Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, 1971.

[Ley04] John Leyden. US wins David Blunkett Lifetime Menace Award (29/07/04). *The Register*, 2004. See `http://www.theregister.co.uk/2004/07/29/big_brother_awards/`.

[LGF03] Ninghui Li, Benjamin Grosof, and Joan Feigenbaum. Delegation Logic: a logic-based approach to distributed authorization. *ACM Transactions on Information and System Security*, pages 128–171, 2003.

[LM03] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proc. PADL*, pages 58–73, 2003 2003.

[LMW02] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[LS97] Emil C. Lupu and Morris Sloman. Reconciling role-based management and role-based access control. In *ACM Workshop on Role-based Access Control*, pages 135–141, 1997.

[LT04] Ninghui Li and Mahesh V. Tripunitara. Security analysis in role-based access control. In *Proc. SACMAT '04*, pages 126–135. ACM Press, 2004.

[LWM01] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In *ACM Conference on Computer and Communications Security*, pages 156–165, 2001.

[Mat70] Yury Matiyasevich. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR*, 191:279–282, 1970.

[Med05] Medix UK plc. Survey (q647) of doctors' views about the National Programme for IT (NPfIT). February 2005.

[MMH02] Lik Mui, Mojdeh Mohtashemi, and Ari Halberstadt. Notions of reputation in multi-agents systems: a review. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 280–287, New York, NY, USA, 2002. ACM Press.

[Mof98] Jonathan D. Moffett. Control principles and role hierarchies. In *Proceedings of the third ACM workshop on Role-based access control*, pages 63–69. ACM Press, 1998.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the sixteenth international conference on Very large databases*, pages 264–277. Morgan Kaufmann Publishers Inc., 1990.

[MS93] Kim Marriott and Harald Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Lett. Program. Lang. Syst.*, 2(1-4):181–196, 1993.

[Mul05] Helene Mulholland. Patients fear online records threaten confidentiality (29/06/05). *The Guardian*, 2005.

[Nat97] National Health Service, UK. The Caldicott committee: report on the review of patient-identifiable information. 1997.

[Nat98] National Health Service, UK. Information for health: an information strategy for the modern NHS 1998-2005. 1998.

[Nat02] National Health Service, UK. ERDIP evaluation: Technical options for the implementation of electronic health record nationally. 2002.

[Nat03] National Health Service, UK. Integrated Care Records Service: Output based specification version 2. 2003.

[Nat05] National Audit Office, UK. Patient choice at the point of GP referral (19/01/05). 2005.

[NC00] SangYeob Na and SuhHyun Cheon. Role delegation in role-based access control. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 39–44. ACM Press, 2000.

[NK03] Mogens Nielsen and Karl Krukow. Towards a formal notion of trust. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 4–7, New York, NY, USA, 2003. ACM Press.

[NO93] Matunda Nyanchama and Sylvia L. Osborn. Role-based security, object oriented databases & separation of duty. *SIGMOD Record*, 22(4):45–51, 1993.

[NO99] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):3–33, 1999.

[Osw05] Malcolm Oswald. Information governance in the NCRS – material for NHS presentations – DRAFT v.05. *National Programme for Information Technology*, January 2005.

[Pal03] Maldwyn Palmer. A complex operation for the NHS spine (14/08/03). *Computer Weekly*, 2003. See `http://www.computerweekly.com/Article124110.htm`.

[Pau94] Lawrence C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828, 1994.

[PP97] Leszek Pacholski and Andreas Podelski. Set constraints: A pearl in research on constraints. In *Principles and Practice of Constraint Programming*, pages 549–562, 1997.

[PW76] Mike S. Paterson and Mark N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on theory of computing*, pages 181–186, 1976.

[Rev93] Peter Revesz. A closed-form evaluation for Datalog queries with integer (gap)-order constraints. *Theoretical Computer Science*, 116(1):117–149, 1993.

[Rev95] Peter Revesz. Datalog queries of set constraint databases. In *Proceedings of the 5th International Conference on Database Theory*, pages 425–438. Springer-Verlag, 1995.

[Rev02] Peter Revesz. *Introduction to constraint databases*. Springer Verlag, 2002.

[Riv98] Ronald L. Rivest. Can we eliminate certificate revocations lists? In *Financial Cryptography*, pages 178–183, 1998.

[RL96] Ronald L. Rivest and Butler Lampson. SDSI – A simple distributed security infrastructure, August 1996. See `http://theory.lcs.mit.edu/~rivest/sdsi10.ps`.

[Rog03] James Rogers. GPs voice patient confidentiality concerns (20/05/03). *Computer Weekly*, 2003. See `http://www.computerweekly.com/Article121897.htm`.

[RZFG01] Carlos Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. SPL: An access control language for security policies with complex constraints. In *Network and Distributed System Security Symposium*, 2001.

[San93] Ravi Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[San98] Ravi Sandhu. Role activation hierarchies. In *Proceedings of the third ACM workshop on Role-based access control*, pages 33–40. ACM Press, 1998.

[SBC+97] Ravi Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Canta, and Charles Youman. The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In *Proceedings of 2nd ACM Workshop on Role-Based Access Control*, pages 41–54, 1997.

[SCFY96] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, pages 47–63. ACM Press, 2000.

[SMW93] Ann Sommerville, Natalie-Jane Macdonald, and R. Weston. *Medical Ethics Today: Its Practice and Philosophy*. British Medical Association, BMJ Publishing Group, 1993.

[SR91] Sundararajar Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 501–511. Morgan Kaufmann Publishers Inc., 1991.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 36, pages 1278–1308, 1975.

[ST03] Vitaly Shmatikov and Carolyn Talcott. Reputation-based trust management. In *Workshop on Issues in the Theory of Security (WITS'03)*, 2003.

[SZ97] Richard T. Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *IEEE Computer Security Foundations Workshop*, pages 183–194, 1997.

[Tom95] David Toman. Top-down beats bottom-up for constraint extensions of datalog. In *Logic Programming*, pages 98–112. MIT Press, 1995.

[Tom97] David Toman. Memoing evaluation for constraint extensions of datalog. *Constraints*, 2(3/4):337–359, 1997.

[Van92] Allen Van Gelder. The well-founded semantics of aggregation. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 127–138. ACM Press, 1992.

[WL02] William H. Winsborough and Ninghui Li. Towards practical automated trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 92–103, 2002.

[WL04] Vicky Weissman and Carl Lagoze. Towards a policy language for humans and computers. In *ECDL*, pages 513–525, 2004.

[WSJ00] William H. Winsborough, Kent E. Seamons, and Vicky E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume 1, pages 88–102, 2000.

[YMB02] Walt Yao, Ken Moody, and Jean Bacon. A model of OASIS role-based access control and its support of active security. *ACM Transactions on Information and System Security*, 5(4), 2002.

[YWS01] Ting Yu, Marianne Winslett, and Kent E. Seamons. Interoperable strategies in automated trust negotiation. In *ACM Conference on Computer and Communications Security*, pages 146–155, 2001.

[ZAC03] Longhua Zhang, Gail-Joon Ahn, and Bei-Tseng Chu. A rule-based framework for role-based delegation and revocation. *ACM Transactions on Information and System Security*, 6(3):404–441, 2003.