A Certified Core Policy Language

Bahman Sistany

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements for the degree of Doctor of Philosophy in
Computer Science

Ottawa-Carleton Institute for Computer Science
School of Information Technology and Engineering
University of Ottawa,
Ottawa, Canada

Abstract

We present the design and implementation of a Certified Core Policy Language (ACCPL) that can be used to express access-control rules and policies. Although full-blown access-control policy languages such as eXtensible Access Control Markup Language (XACML) [OAS13] already exist, because access rules in such languages are often expressed in a declarative manner using fragments of a natural language like English, it isn't always clear what the intended behaviour of the system encoded in these access rules should be. To remedy this ambiguity, formal specification of how an access-control mechanism should behave, is typically given in some sort of logic, often a subset of first order logic. To show that an access-control system actually behaves correctly with respect to its specification, proofs are needed, however the proofs that are often presented in the literature are hard or impossible to formally verify. The verification difficulty is partly due to the fact that the language used to do the proofs while mathematical in nature, utilizes intuitive justifications to derive the proofs. Intuitive language in proofs means that the proofs could be incomplete and/or contain subtle errors.

ACCPL is small by design. By small we refer to the size of the language; the syntax, auxiliary definitions and the semantics of ACCPL only take a few pages to describe. This compactness allows us to concentrate on the main goal of this thesis which is the ability to reason about the policies written in ACCPL with respect to specific questions. By making the language compact, we have stayed away from completeness and expressive power in several directions. For example, ACCPL uses only a single policy combinator, the conjunction policy combinator. The design of ACCPL is therefore a trade-off between ease of formal proof of correctness and expressive power.

We also consider ACCPL a core policy access-control language since we have retained the core features of many access-control policy languages. For instance ACCPL employs a single condition type called a “prerequisite” where other languages may have very expressive and rich sets of conditions.

Acknowledgements

I would like to thank my thesis advisor, Amy Felty, for her support and encouragement, and her constant feedback and guidance during the course of my PhD studies.

Dedication

This thesis is dedicated to my wife Sanam who stood by me and kept me going when life threatened to get in the way, and to my wonderful sons Roitan and Rod Barzin for their sacrifice and patience during the course of my PhD studies.

I also dedicate this work to my parents, Shamseddin and Robab, who first taught me the value of education and for their continued encouragement and support throughout my studies.

Table of Contents

Listings

Chapter 1

Introduction

In this thesis, we describe the design of a Certified Core Policy Language (ACCPL) and its implementation in the Coq Proof Assistant. Using Coq to implement ACCPL as it was being defined was an important factor in its design, allowing us to address the trade-off between expressive power and ease of formal proof of correctness. We argue that ACCPL is significantly more amenable to analysis and reasoning than other access-control policy languages, while still retaining sufficient expressive power, and we describe how the design choices that were made contributed to this ease of reasoning.

The semantics of ACCPL are specified by translation from policy statements together with an access request and an environment containing all relevant facts, to decisions. We present results showing the translation functions behave correctly with respect to the decision question that asks whether a request to access a resource may be granted or denied, given a policy. The translation functions also cover the case where a given policy does not apply to a request in which case a decision of non-applicable is rendered. Our results show that for each access request, the translation algorithm terminates on all input policies with a decision of granted, denied or non-applicable. Proving that the translation algorithm terminates on all input policies (given an access request) with a decision of granted, denied or non-applicable, is the specific goal based on which the semantics were defined. We further capture this property as a specification (expressed as an inductive predicate in Coq) and show how the translation algorithm meets this specification for a given policy.

To motivate the design of ACCPL, we need to go back to the origins of access-control and review the progression of the models from the most basic and perhaps oldest to the model ACCPL implements. We will start the review of access-control models based on the survey done in [NIS09], in Section 1.1, but first the definition of “access-control” is in order.

Authorization refers to the process of rendering a decision to allow access to a resource or asset of interest. By the same token all unauthorized access requests to resources must be controlled and ultimately denied, hence the term “access-control”.

1.1 Access Control Models

Access Control List (ACL) is perhaps the oldest and the most basic access-control model. A list of subjects along with their rights are kept per resource or object of interest. Every time a subject makes an access request on an object, the ACL of the object is consulted and access is either granted or denied to the requester based on whether or not the requester is listed in the ACL and has the correct set of rights.

Capabilities based access-control works based on a list of objects and associated rights. The list of objects and the associated rights comprise an “unforgeable” ticket that a reference monitor checks to allow access (or not). Capabilities based systems don’t need to authenticate users as ACL based systems do.

In Role-based Access Control (RBAC) systems a requester’s role determines whether access is granted or denied. In this model, users belong to roles and rights are associated with roles so no direct association between users and rights exist. Roles are meant to group users and add flexibility when assigning rights. RBAC systems naturally solve the problem of assigning ACLs for a large group of users and manage the administration cost of changing users’ rights in ACL based systems.

Despite many advantages of RBAC systems, some disadvantages also exist. Often a role needs to be decomposed into sub-roles based on the type of resource to be administered, and perhaps also based on the location that the resource serves. Basically RBAC suffers from a lack of a sub-typing mechanism whereby individual members of a group/role may be differentiated and access is granted or denied based on a more granular set of attributes. The Attribute-based Access Control (ABAC) model was proposed to fulfill this granularity requirement. In ABAC access control decisions are made based on a set of attributes, associated with the subject making the request, the environment, and/or the resource itself.

In order to harmonize access control in large environments with many subjects and objects and disparate attributes, Policy-based Access Control (PBAC) model has been proposed. PBAC allows for a more uniform access-control model across the system. PBAC systems help create and enforce policies that define who should have access to what resources, and under what circumstances. There is also a need for large organizations to put in place mechanisms such that access-control rules can be easily audited. This calls for a data-driven approach to access-control where the data, in this case the access-control rules, are available to read and analyze. A data-driven approach like PBAC additionally helps with modularity of the system as changes in access control rules will have almost no impact to the underlying system the rules are meant to protect.

Because of the cited advantages of PBAC and its generality and wide spread use, PBAC is the model ACCPL implements.

1.2 A Core Policy Language for PBAC Systems

Does there already exist a suitable policy language that can be used for expressing general access-control expressions? How about eXtensible Access Control Markup Language (XACML) [OAS13], Open Digital Rights Language (ODRL) [Ian02] or eXtensible rights Markup Language (XrML) [WLD$^+$02]?

Currently the most popular Rights Expression Languages (REL)s are XrML, and ODRL. Both of these languages are XML based and are considered declarative languages. XrML has been selected by the the Moving Picture Experts Group, MPEG, to be the REL for *MPEG-21* which is an ISO standard for multimedia applications. ODRL is also a standards-based REL which has been accepted as part of the W3C community with the mandate of standardizing how rights and policies, related to the usage of digital content on the Open Web Platform, *OWP* [Wik15b], are expressed. ODRL 2.0 supports expression of rights and also privacy rules for social media while ODRL 1.0 was only dealing with the mobile ecosystem – ODRL 1.0 was adopted by the Open Mobile Alliance, *OMA* in 2000.

RELs, or more precisely Digital Rights Expression Languages (DREL)s when dealing with digital assets deal with the "rights definition" aspect of the Digital Rights Management (DRM) ecosystem. A DREL, allows the expression and definition of digital asset usage rights such that other areas of the DRM ecosystem, namely the enforcement mechanism and the usage tracking components can function correctly.

As popular as both XrML and ODRL are, their adoption and usage is still somewhat limited in practice. Both Apple and Microsoft for example have defined their own lightweight RELs [JHM06] in FairPlay [Wik15a] and in PlayReady [Wik15c]. The authors of [JHM06] argue that both these RELs (XrML and ODRL) and others are simply too complex to be used effectively (for expressing rights) since they also try to cover much of the the enforcement and tracking aspects of DRMs.

DRM refers to the digital management of rights associated with the access or usage of digital assets. There are various aspects of rights management however. According to the authors of the white paper "A digital rights management ecosystem model for the education community" [CPR04] digital rights management systems cover the following four areas:

1. Defining rights
2. Distributing/Acquiring rights
3. Enforcing rights
4. Tracking usage

XACML is a high-level and platform independent access control system that is also XML based. XACML is an OASIS standard which defines a language for the definition of policies and access requests, and a workflow to achieve policy enforcement [MPT12]. According to Masi et al [MPT12] designing XACML access control policies is difficult and error prone. Furthermore XACML comes without a formal semantics as do both XrML and ODRL. The XACML standard is written in prose and contains quite a number of loose points that may give rise to different interpretations and lead to different implementation choices [MPT12].

XACML, ODRL and XrML are all PBAC based languages where ODRL and XrML differ from XACML by their focus on digital assets protection and in general DRM, hence the term REL. All three are full-blown and custom languages that have one thing in common; they suffer from a lack of formal semantics. Additionally all of these languages cover much more than policy expressions leading to access decisions; they also address enforcement of policies (ODRL and XrML specifically and DRM in general distinguish themselves from general access-control languages by additionally addressing enforcement of policies beyond where the policies were generated). A third reason that made these custom languages unsuitable as a core policy language was the fact that they are limited in terms of what can be built on top of them; for example expressing hierarchical role-based access-control in XACML requires a fairly complex encoding [TK06].

Rights expressions in DRELs and specifically ODRL are used to arbitrate access to assets under conditions. The main construct in ODRL is the "agreement" which specifies users, asset(s) and policies whereby controls on users' access to the assets are described. This is very similar to how access control conditions are expressed in access control policy languages such as XACML [ANP$^+$03] and SELinux [SVLS02]. In fact several authors have worked on interoperability between RELs and access control policy languages, specifically between ODRL and XACML [PRD05, MRD09] and also on translation from high level policies of XACML to low-level and fine grained policies of SELinux [ASLZ08].

A policy language that was designed with logic and formal semantics and also one that was small and extendible was clearly needed. We started by looking at Lithium [HW08b] and subsequently Pucella and Weissman's subset of ODRL [PW06] as potential core languages. Lithium uses DRM as the main application of its access-control system whereas Pucella and Weissman's primary goal is to define formal semantics for ODRL whose main application in turn is DRM. We will use Pucella and Weissman's subset of ODRL as the basis for ACCPL and in doing so treat digital rights as our main access-control application without loss of generality with respect to other applications, with the final goal of performing formal verification on policies written in ACCPL.

1.3 Formal Semantics for PBAC Languages

Formal methods help ensure that a system behaves correctly with respect to a specification of its desired behaviour [Pie02]. This specification of the desired behaviour is what's referred to as *semantics* of the system. Using formal methods requires defining precise and formal semantics, without which analysis and reasoning about properties of the system in question would become impossible. For example, an issue with the current batch of RELs are due to their semantics being expressed in a natural language (e.g. English) which by necessity results in ambiguous and open to interpretation behaviour.

To formalize the semantics of PBAC languages several approaches have been attempted by various authors. Most are logic based [HW08b, PW06] while others are based on finite-automata [HKS04], operational semantics based interpreters [SS09a] and web ontology (from the Knowledge Representation Field) [KG10].

1.4 Logic Based Semantics

Formal logic can represent the statements and facts we express in a natural language like English. Propositional logic is expressive enough to express simple facts as propositions and uses connectives to allow for the negation, conjunction and disjunction of the facts. In addition simple facts can be expressed conditionally using the implication connective. Propositional logic however is not expressive enough to express policies of the kind used in languages like ODRL and XrML. For example, a simple policy expressed in English like “All who pay 5 dollars can watch the movie Toy Story” cannot be expressed in propositional logic because the concept of quantifiers doesn’t exist in propositional logic.

A richer logic such as “First Order Logic” (*FOL*), is more suitable and has the expressive power to represent policies written in English. Moreover, FOL can be used to capture the meaning of policies in an unambiguous way.

Halpern and Weissman [HW08b] propose a fragment of FOL to represent and reason about policies. The fragment of FOL they arrive at is called *Lithium* which is decidable and allows for efficiently answering interesting queries. Lithium restricts policies to be written, in part based on the concept of “bipolarity” which disallows by construction policies that both permit and deny an action on an object. Pucella and Weissman [PW06] specify a predicate logic based language that represents a subset of ODRL.

1.5 Specific Problem

Policy languages and the agreements written in those languages are meant to implement specific goals such as limiting access to specific assets. The tension in designing a policy language is usually between how to make the language expressive enough, such that the design goals for the policy language may be expressed, and how to make the policies verifiable with respect to the stated goals.

As stated earlier, an important part of fulfilling the verifiability goal is to have formal semantics defined for policy languages. For ODRL, authors of [PW06] have defined a formal semantics based on which they declare and prove a number of important theorems (their main focus is on stating and proving algorithm complexity results). However as with many paper-proofs, the language used to do the proofs while mathematical in nature, uses many intuitive justifications to show the proofs. As such these proofs are difficult to verify or more importantly to “derive”. Furthermore the proofs can not be used directly to render a decision on a sample policy (e.g. whether to allow or deny access to an asset). Of course one may (carefully) construct a program based on these proofs for practical purposes but certifying such programs correct presents additional verification challenges, even assuming the original proofs were in fact correct.

1.6 Contributions

In this thesis we have built a language called ACCPL based on ODRL and starting with definitions in [PW06]. The ACCPL framework has been encoded in *Coq* [BC04] which is both a programming language and a proof-assistant. We have specified and proved ACCPL correct with respect to properties of interest (see section 6.1) in Coq which will allow us to extract programs from the proofs; the executable programs can be used on specific policies and a query, to render a specific decision such as "a permission has been granted".

We originally started with a specific subset of [PW06] so that we could concentrate on what we believed to be the essence or core of the language. For example, we started with only one of three different kinds of "facts" (that may affect the permit/deny type decisions). We also had to change some of the language productions to allow for Coq's requirement for clearly terminating recursion. Initially we intended to maintain the central semantic definitions including "Closed World Assumptions" [PW06] where the semantics only specify explicitly Permitted and NotPermitted answers however we discovered the semantics as stated by Pucella and Weissman [PW06] are not explicit and therefore the decision question that asks whether a request to access a resource may be granted or denied, may not be answered in all cases. We have therefore made major modifications to the semantics of Pucella and Weissman's language such that an answer to a request for access to a resource may be determined unambiguously and for all cases.

Our results subsume an important sub-category, namely inconsistency or conflict-detection in policy expressions or rules. St-Martin and Felty [SF16] describe and implement in Coq a conflict detection algorithm for detecting conflicts in XACML access control rules. XACML is an expressive and at the same time complex policy language which makes conflict detection a difficult task. The authors of [SF16] then prove the conflict detection algorithm correct (or certified) by developing a formal proof in Coq. The proof is rather complex and involves a large number of cases, including many corner cases that were difficult to get right [SF16]. For ACCPL we have formally proven that conflicts are not possible.

Given that ACCPL is a core policy language with semantics that have been certified correct, we could use ACCPL to implement various (more expressive) policy languages. In addition ACCPL could be used as an intermediate language to reason about interoperability between those policy languages [PRD05,MRD09]. In this manner our language ACCPL can be viewed as a extendable language, complete with defined and verified semantics, that can be used as the basis for implementing various policy languages with more expressive power (e.g. W3C's ODRL and SELinux).

For access to the Coq source code for ACCPL, please refer to `http://www.site.uottawa.ca/~afelty/accpl/`.

Chapter 2

Examples

We will start this chapter by presenting an example policy statement expressed using ODRL's XML-based syntax. The statement "the asset TheReport may be printed a total of 2 times by Alice only" is illustrated in listing 2.1.

Listing 2.1: First Agreement for Alice and Bob in XML

```
<agreement>
 <asset> <context> <uid> The Report </uid> </context> </asset>
 <permission>
   <print>
    <constraint> <name> Alice </name> </constraint>
    <constraint> <count> 2 </count> </constraint>
   </print>
 </permission>
 <party> <context> <name> Alice </name> </context> </party>
 <party> <context> <name> Bob </name> </context> </party>
</agreement>
```

The statement in listing 2.1 is shown in listing 2.2 using the abstract syntax from [PW06].

Listing 2.2: First Agreement for Alice and Bob

```
agreement
 for Alice and Bob
 about The Report
 with True -> and[Alice, count[2]] =>$_{id1}$ print.
```

A similar but slightly more complex (than the one in 2.1) example is discussed in [PW06] and we list it here (see listing 2.3). This particular policy statement is not expressible in ACCPL due to lack of support for two kinds of conditions. One kind is the "prePay[n]" condition supported by Pucella and Weissman [PW06]'s fragment of ODRL (as a prerequisite). The second kind mentioned in this statement, "Mary's computer", is some kind of location based condition that the authors of [PW06] refer to, but we don't find it defined anywhere.

As we discuss later in this thesis (see sections 7.1 and 7.2) both of these prerequisites (and others) could potentially be implemented to extend expressibility of ACCPL.

Listing 2.3: Agreement for Alice and Bob in XML [PW06]

```
<agreement>
 <asset> <context> <uid> Treasure Island </uid> </context>
    </asset>
 <permission>
   <display>
    <constraint> <name> Mary's computer </name> </
       constraint>
   </display>
   <print>
    <constraint> <count> 2 </count> </constraint>
   </print>
 </permission>
 <party> <context> <name> Mary </name> </context> </party>
</agreement>
```

The statement in listing 2.3 is shown in listing 2.4 using the abstract syntax form.

Listing 2.4: Agreement for Alice and Bob [PW06]

```
agreement
 for Mary
 about Treasure Island
 with prePay[5.00] -> and[[Mary's computer] =>$_{id1}$ display,
                                  count[2] =>$_{id2}$ print.
```

Chapter 3

ACCPL Syntax

3.1 Introduction

We follow the style of [PW06] by using abstract syntax to express policy statements in ACCPL. Abstract syntax is a more compact representation than XML which is what all the XML-based policy languages such as ODRL use. Furthermore abstract syntax simplifies specifying the semantics as we shall see later.

In the following we will cover the abstract syntax of ACCPL that we later implement in Coq using constructs such as inductive types and function definitions.

3.2 Environmental Facts

To determine the outcome of policies, specified conditions in those policies are evaluated but to do so environmental facts are often needed. In the DRM realm with its focus on usage control, certain facts are typically tracked in the environment. The count of how many times an asset has been accessed, the amount a user has paid to access an asset and finally whether a user has made an attribution (e.g. mentioning the content owner by name) are examples of the kind of facts environments hold.

In ACCPL, agreements and facts (i.e. environments) will refer to a count of how many times each policy should be used and has been used respectively, to justify an action. This is the only fact that ACCPL will cover although we conjecture adding other simple facts such as attributions and "amounts paid" and the machinery to support those facts should not change the verification goals for ACCPL and the established results in a significant way. We will defer to later in the thesis (see section 7.1), the discussion of how and whether more general and complex facts in the environment will impact ACCPL and our verification results.

We will describe ACCPL in a *BNF* grammar that looks more like Pucella and Weissman's subset grammar [PW06]. BNF style grammars are more abstract as they only give suggestions about the surface syntax of expressions without getting into lexical analysis and parsing related aspects such as precedence order of operators [PCG+11]. The Coq version in contrast is the encodings of the BNF definitions which is typically used for building compilers and interpreters. We will present both the BNF definition and its Coq encoding for each construct of ACCPL.

3.3 Productions

The top level ACCPL production is the `<agreement>`. An agreement expresses what actions a set of subjects may perform on an object and under what conditions. Syntactically an agreement is composed of a set of subjects called a principal or `<prin>`, an `<asset>` and a `<policySet>` (see listing 3.1).

Listing 3.1: agreement

```
<agreement> ::=
     'agreement' 'for' <prin> 'about' <asset> 'with' <
        policySet>
```

Principals (`<prin>`) are composed of `<subject>`s which are specified based on the application e.g. `Alice`, `Bob`, etc (see listing 3.2).

Listing 3.2: prin

```
<prin> ::=  { <subject_1>, ..., <subject_m> }
```

Assets are also application specific and we will continue using specific ones for the DRM application (taken from [PW06]). `TheReport`, `ebook`, and `latestJingle` are examples of specific assets we will be using throughout (see listing 3.3).

Listing 3.3: asset

```
<asset> ::= TheReport | ebook | latestJingle | ...
```

Subjects are treated similarly to assets. We will use specific subjects like `Alice` and `Bob` (see listing 3.4).

Listing 3.4: subject

```
<subject> ::= Alice | Bob | ...
```

Agreements express who may perform an action on an asset. They include a set of subjects (in the form of a principal), an asset and a policy set. A policy set is a primitive policy set implying non-nested policy sets. Note that we could define various combining operators for policy sets such as conjunctions and disjunctions but we keep ACCPL's policy sets limited to the primitive kind and their conjunctions. We do however, use a combining operator when it comes to dealing with policies later on. Each primitive policy set specifies a `<prerequisite>` and a `<policy>`. Intuitively if the prerequisite "holds" the policy is taken into consideration. Otherwise the policy will not be looked at (see listing 3.5).

Listing 3.5: policySet and primPolicySet

```
<policySet> ::=
   <primPolicySet>                ; primitive policy set

<primPolicySet> ::=
    <primInclusivePolicySet>     ; primitive inclusive policy set
 |  <primExclusivePolicySet>     ; primitive exclusive policy set
```

Some primitive policy sets are specified as inclusive as opposed to others that are explicitly specified as exclusive. Primitive exclusive policy sets are exclusive to an agreement's users in that only those users may perform the actions specified in the policy set. The implication is that all other users who are not specified in the agreement's principal are forbidden from performing the specified actions, no matter whether the prerequisite holds or not (see listing 3.6).

Listing 3.6: primExclusivePolicySet

```
<primExclusivePolicySet> ::=
   <prerequisite> ↦ <policy>       ; primitive exclusive policy set
```

Not surprisingly we also define primitive inclusive policy sets that don't enforce any exclusivity to the agreement's users (see listing 3.7).

Listing 3.7: primInclusivePolicySet

```
<primInclusivePolicySet> ::=
   <prerequisite> → <policy>       ; primitive inclusive policy set
```

A primitive policy specifies an action to be performed on an asset, depending on whether the policy's prerequisite holds or not. If the prerequisite holds the agreement's user is permitted to perform the action on the agreement's asset; otherwise permission is denied. Pucella and Weissman's subset of ODRL [PW06] specifies a unique identifier for each policy to help the translation (from agreements to formulas). ACCPL has maintained the policy identifier and we include it here in our definition of the policy construct, however, as far as the proofs are concerned the policy identifier could be removed without a loss to the obtained results (see listing 3.8).

Listing 3.8: primPolicy

```
<primPolicy> ::=
   <prerequisite> ⇒<policyId> <act>
      ; primitive policy
```

A policy is made up of primitive policies. Primitive policies are grouped together using the conjunction combining operator (see listing 3.9).

Listing 3.9: policy

```
<policy> ::=
 'and'[ <primPolicy_1>, ...,
                 <primPolicy_m> ]
                    ; conjunction
```

Similar to assets and subjects, actions are application specific and we use specific ones taken from [PW06] such as `Display` and `Print` (see listing 3.10).

Listing 3.10: act

```
<act> ::= Play | Print | Display | ...
```

A policy identifier (`<policyId>`) is a unique identifier specified as (increasing) positive integers (see listing 3.11).

Listing 3.11: policyId

```
<policyId> ::= N
```

In ACCPL a `<primPrerequisite>` is either True or it is a `<constraint>`. The True prerequisite always holds. A constraint is an intrinsic part of a policy and cannot be influenced by agreement's users. Minimum height requirements for popular attractions and rides are examples of what we would consider a constraint (although not supported in ACCPL). A constraint can also be negative, specified by the keyword 'not' in front of `<constraint>`. A `<prerequisite>` is a set of primitive prerequisites which are closed under conjunction operator specified by the keyword 'and' in front of the list of the primitive prerequisites separated by commas (see listing 3.12).

Listing 3.12: prerequisite

```
<primPrerequisite> ::=
       'True'                        ; always true
    |  <constraint>                  ; constraint
    |  'not' [ <constraint> ]        ; suspending constraint

<prerequisite> ::=
    |  'and'[ <primPrerequisite_1>, ..., <primPrerequisite_m> ]
       ; conjunction
```

Constraints are either of the principal kind, the count kind, or the count by principle kind. Principal constraints require matching to the users listed following the keyword `<prin>`. For example, the constraint of "the user being Alice" is a constraint of type principal. A count constraint refers to the number of times the user of an agreement has invoked policies to justify her actions. If the count constraint is part of a policy then the count refers to that single policy. In the case that the count constraint is part of a policy set or if the policy is a conjunction, then the count refers to the set of policies specified in the policy set or in the policy conjunction as the case may be (see listing 3.13).

Listing 3.13: constraint

```
<constraint> ::=
       <prin>                   ; principal
    |  ’Count’ [N]              ; number of executions
    |  <prin> (’Count’ [N])     ; number of executions by prin
```

Chapter 4

ACCPL Syntax in Coq

4.1 Introduction to Coq

Coq is known first and foremost as a proof-assistant. The underlying formal language that Coq uses is a much more expressive version of typed lambda calculus called Calculus of (Co)Inductive Constructions (CIC) where proofs and programs can both be represented. For example, CIC adds polymorphism (terms depending on types), type operators (types depending on types) and dependent types (types depending on terms).

Specifications of programs in Coq may be expressed using the specification language *Gallina* [Hue92]. Coq is then used to develop proofs to show that a program's run-time behaviour satisfies its specification. Such programs are called *certified* because they are formally verified and confirmed to conform to their specifications [BC04].

Assertions or propositions are statements about values in Coq such as `3<8` or `8<3` that may be true, false or even be only conjectures. To verify that a proposition is true a proof needs to constructed. While paper-proofs use a combination of mathematics and natural language to describe their proofs, Coq provides a formal (and therefore unambiguous) language that is based on proof-theory to develop proofs in. Verification of complex proofs is possible because one can verify the intermediate proofs or sub-goals in steps, each step being derived from the previous by following precise derivation rules. The Coq proof engine solves successive goals by using predefined *tactics*. Coq tactics are commands to manipulate the local context and to decompose a goal into simpler goals or sub-goals [BC04].

4.2 ACCPL Syntax

ACCPL productions were presented as high level abstract syntax in Section 3.3 of chapter 3. Below we present the corresponding encodings in Coq.

An `agreement` is a new inductive type in Coq by the same name. The constructor `Agreement` takes a `prin`, an `asset` and a `policySet`. `prin` is defined to be a nonempty list of `subjects` (see listing 4.1).

ACCPL types asset, subject, act and policyId are base types and are simply defined as nat which is the datatype of natural numbers defined in coq's library module Coq.Init.Datatypes (nat is itself an inductive datatype). We use Coq constants to refer to specific objects of each type. For example, the subject 'Alice' is defined as

```
Definition Alice:subject := 101.
```

and the act 'Play' as

```
Definition Play : act := 301.
```

Note that whereas in the implementation of asset, subject and act in this chapter we used nats, in the abstract syntax chapter (chapter 3) in the corresponding listings 3.3, 3.4 and 3.10 we used syntactic tokens like 'Alice' and 'ebook' directly.

Next we define the policySet datatype which is the direct implementation of the abstract syntax presented in listing 3.5. A policySet is constructed only one way: by calling the PPS constructor which takes a primPolicySet as input. There are two ways a primPolicySet can be constructed (see listing 3.5 for the abstract syntax version) corresponding to two constructors: PIPS and PEPS.

PIPS takes a primInclusivePolicySet as input while PEPS takes a primExclusivePolicySet. Both primInclusivePolicySet and primExclusivePolicySet types are constructed by taking a preRequisite and a policy as parameters (see listing 3.7 and 3.6 for the abstract syntax versions).

A policy is defined as a datatype with the constructor Policy which takes a nonempty list of primitive policies, or primPolicys. A primPolicy is constructed by calling PrimitivePolicy which takes a prerequisite, preRequisite, a policy identifier, policyId, and an action, act (see listing 3.8 for the abstract syntax version). Ignoring the policyId for a moment, a primitive policy consists of a prerequisite and an action. Intuitively if the prerequisite holds, the action is allowed to be performed on the asset.

Listing 4.1: ACCPL: Coq Version of Agreement

```
Inductive agreement : Set :=
 | Agreement : prin → asset → policySet → agreement.

Definition prin := nonemptylist subject.

Definition asset := nat.

Definition subject := nat.

Definition act := nat.

Definition policyId := nat.

Inductive policySet : Set :=
 | PPS : primPolicySet → policySet.

Inductive primPolicySet : Set :=
 | PIPS : primInclusivePolicySet → primPolicySet
 | PEPS : primExclusivePolicySet → primPolicySet.

Inductive primInclusivePolicySet : Set :=
 | PrimitiveInclusivePolicySet : preRequisite → policy →
   primInclusivePolicySet.

Inductive primExclusivePolicySet : Set :=
 | PrimitiveExclusivePolicySet : preRequisite → policy  →
   primExclusivePolicySet.

Inductive policy : Set :=
 | Policy : nonemptylist primPolicy → policy.

Inductive primPolicy : Set :=
 | PrimitivePolicy : preRequisite → policyId → act → primPolicy.
```

The data type `nonemptylist` reflects the definition of “policy conjunction” (see the definition of `nonemptylist` in listing 4.2). Essentially `nonemptylist` represents a list data structure that has at least one element and it is defined as a new *polymorphic* inductive type in its own Coq section. See the abstract syntax version for “policy conjunction” in listing 3.9.

Listing 4.2: nonemptylist

```
Section nonemptylist.

Variable X : Set.

Inductive nonemptylist : Set :=
  | Single : X → nonemptylist
  | NewList : X → nonemptylist → nonemptylist.

End nonemptylist.
```

In listing 4.3 a `preRequisite` is defined as a datatype with the constructor `PreRequisite` which takes a nonempty list of primitive prerequisites, or `primPreRequisite`s. A `primPreRequisite` is defined as a new datatype with constructors `TruePrq`, `Constraint` and `NotCons` (see listing 3.12 for the abstract syntax equivalent). `TruePrq` represents the always true prerequisite. The `Constraint` constructor takes a value of type `constraint` which we will describe below. Intuitively a constraint is a prerequisite to be satisfied that is outside the control of the user(s). For example, the constraint of being ‘Alice’. The constructor `NotCons` is defined the same way the `Constraint` constructor is. This constructor is defined as the type `constraint` and it is meant to represent the negation of a `constraint` as we shall see in the translation (see listing 5.23). In listing 4.3 we also show `makePreRequisite` which is a convenience function that given a `primPreRequisite`, builds a value of type `preRequisite`.

Listing 4.3: preRequisite

```
Inductive primPreRequisite : Set :=
  | TruePrq : primPreRequisite
  | Constraint : constraint → primPreRequisite
  | NotCons : constraint → primPreRequisite.

Inductive preRequisite : Set :=
  | PreRequisite : nonemptylist primPreRequisite → preRequisite.

Definition makePreRequisite (prq:primPreRequisite) : preRequisite :=
 (PreRequisite (Single prq)).
```

Finally a `constraint` (see listing 3.13 for the abstract syntax equivalent) is defined as a new datatype with constructors `Principal`, `Count` and `CountByPrin`. See listing 4.4 for the definition of `constraint` and examples of how the three different kinds of constraints are constructed.

Listing 4.4: Constraint Definition and the Three Kinds of Constraints

```
Inductive constraint : Set :=
  | Principal : prin  → constraint
  | Count : nat → constraint
  | CountByPrin : prin → nat → constraint.

(Constraint (Principal (Single Alice))).

(Constraint (Count 10)).

(Constraint (CountByPrin (Single Alice) 10)).
```

We started with the encoding of the policy statement for "First Agreement for Alice and Bob" in chapter 3 but we deferred the definition of the Coq constructs used to encode that agreement. All the definitions needed to encode the agreement for Alice and Bob have now been defined so we now show how that agreement looks like as encodings of ACCPL constructs in Coq in the listing 4.5. Recall the listing 2.2 for the abstract syntax form of the statement.

We show how the "First Agreement for Alice and Bob" statement would be expressed in ACCPL using Coq constructs, in listing 4.5.

Listing 4.5: Expressing First Agreement for Alice and Bob in ACCPL

```
Definition ps_xml_p1prq1:primPreRequisite :=
  (Constraint (Principal (Single Alice))).

Definition ps_xml_p1prq2:primPreRequisite :=
  (Constraint (Count 2)).

Definition ps_xml_prq:preRequisite :=
  (PreRequisite (NewList ps_xml_p1prq1 (Single ps_xml_p1prq2))).

Definition ps_xml_p1:primPolicy :=
  (PrimitivePolicy ps_xml_prq id1 Print).

Definition ps_xml_p:policy :=
  (Policy (Single ps_xml_p1)).

Definition ps_xml:primPolicySet :=
  PIPS (PrimitiveInclusivePolicySet
    (makePreRequisite TruePrq) ps_xml_p).

Definition Axml := Agreement (NewList Alice (Single Bob)) TheReport (PPS ps_xml).
```

We show how the "First Agreement for Alice and Bob" policy statement looks like as an ACCPL construct, in listing 4.6.

Listing 4.6: Fully Built First Agreement for Alice and Bob in ACCPL

```
Agreement (Alice, [Bob]) TheReport
      (PPS
        (PIPS
          (PrimitiveInclusivePolicySet (PreRequisite [TruePrq])
            (Policy
              [PrimitivePolicy
                (PreRequisite
                  (Constraint (Principal [Alice]),
                   [Constraint (Count 2)])) id1 Print]))))
```

Chapter 5

ACCPL Semantics In Coq

We specify the semantics of ACCPL as a translation function from an agreement together with an access request and an environment containing all relevant facts, to decisions (more concretely to a set of `results`–see listing 5.6). Specifying the semantics as translations from an agreement, an access request and an environment to decisions, mostly follows the style Pucella and Weissman [PW06] use to define the semantics for their ODRL fragment. In Pucella and Weissman's language, the translation functions are defined from agreements to formulas in many-sorted First-Order Logic (FOL) with equality. The translation functions for ACCPL return a specific decision based on whether there are proof terms for certain conditions and/or proof terms for the negation of those conditions (see Section 5.1 for more details).

The translation functions plus the auxiliary types and infrastructure which implement the semantics for ACCPL have been encoded in Coq. The Coq code implementing the semantics is high-level enough to be readable as a specification of the semantics. This is one reason why we do not use abstract syntax to describe the semantics as we did for the syntax of ACCPL (see chapter 3). The other reason not to use abstract syntax the way Pucella and Weissman [PW06] do for their semantics, is the difficulty of coming up with abstract syntax that is intuitive and equivalent to the translation functions implemented in Coq and listed in this chapter.

Whether an agreement and the accompanying access request or query is translated into a decision indicating a permission is granted or denied depends on the agreement in question and the specifics of the request, but also on the facts recorded in the environment. For ACCPL those facts revolve around the number of times a policy has been used to justify an action. We encode this information in Coq as a new inductive type representing the environment. An `environment` (see listing 5.1) is a conjunction of equalities of the form `count(s, policyId) = n` which are called "count equalities" (see Section 3.2 for background on environments).

The Coq version of a count equality is a new inductive type called `count_equality`. An environment is defined to be a nonempty list of `count_equality` objects (see listing 5.1). Function `make_count_equality` in listing 5.1 is simply a convenience function that builds `count_equality` objects.

Listing 5.1: Environments and Counts

```
Inductive count_equality : Set :=
  | CountEquality : subject → policyId → nat → count_equality.

Definition make_count_equality
 (s:subject)(id:policyId)(n:nat): count_equality :=
 CountEquality s id n.

Inductive environment : Set :=
 | SingleEnv : count_equality → environment
 | ConsEnv :  count_equality →  environment → environment.
```

For an example of how environments are created see listing 5.2.

Listing 5.2: Defining Environments

```
Definition e1 : environment :=
 (SingleEnv (make_count_equality Alice id1 8)).
```

We also define a `getCount` function that given a pair consisting of a subject and policy id, looks for a corresponding count in the environment. The `getCount` function assumes the given environment is consistent (meaning it won't return two different counts for the same pair of subject and policy id), so it returns the first matched `count` it sees for a `(subject, id)` pair. If a `count` for a `(subject, id)` pair is not found it returns 0.

We have defined the necessary representations in Coq in the form of an inductive predicate (called `env_consistent`) to verify the consistency assumption mentioned above (see listing 5.3). For a given environment `e1`, we would require a proof of the proposition "`env_consistent e1`" before translation begins. Note that we have defined helper lemmas to help complete such proofs. In the listing 5.3, we have also shown the definition of the function `inconsistent` which states that two `count_equality` values are inconsistent for the same pair of subject and policy id, if the counts are different. Another helper definition shown in listing 5.3 is the function `formula_inconsistent_with_env` which captures the proposition "this count formula is inconsistent with this environment".

Listing 5.3: Consistency of Environments

```
Inductive env_consistent : environment → Prop :=
 | consis_1 : ∀ f, env_consistent (SingleEnv f)
 | consis_2 : ∀ f g, ~(inconsistent f g) → env_consistent (ConsEnv f (SingleEnv g))
 | consis_more : ∀ f e,
    env_consistent e → ~(formula_inconsistent_with_env f e) → env_consistent (
     ConsEnv f e).

Definition inconsistent (f1 f2 : count_equality) : Prop :=
  match f1 with (CountEquality s1 id1 n1) ⇒
    match f2 with (CountEquality s2 id2 n2) ⇒
      s1 = s2 → id1 = id2 → n1 <> n2
    end
  end.

Fixpoint formula_inconsistent_with_env (f: count_equality)
        (e : environment) : Prop :=
 match e with
   | SingleEnv g ⇒  inconsistent f g
   | ConsEnv g rest ⇒ (inconsistent f g) \/ (formula_inconsistent_with_env f rest)
 end.
```

5.1 The sumbool Type

The sumbool type is a boolean type defined in the Coq standard library module Coq.Init.Specif. The sumbool type captures the idea of program values that indicate which of two propositions is true [Chl11]. The sumbool type is equipped with the justification of their value [Tea04] which help with proofs. Using a tactic like destruct [Tea04] two subgoals are generated, one for each form of the sumbool instance, however the justifications also show up as hypothesis helping with discharging of the subgoals. The definition of sumbool from Coq library module Coq.Init.Specif is listed in listing 5.4; notice the use of the where clause in this listing, which allows notation to be simultaneously defined (and used) when presenting a new inductive definition.

Listing 5.4: sumbool type

```
Inductive sumbool (A B:Prop) : Set :=
  | left : A → {A} + {B}
  | right : B → {A} + {B}
 where "{ A } + { B }" := (sumbool A B) : type_scope.
```

As an example of how sumbool type is used in the Coq standard library, see the listing 5.5, for the definition of the type sumbool_of_bool. The definition in listing 5.5 and the following description are taken from the library Coq.Bool.Sumbool [Tea04].

> sumbool A B, which is written A+B, is the informative disjunction “A or B”, where A and B are logical propositions. Its extraction is isomorphic to the type of booleans. A boolean is either true or false, and this is decidable.

Listing 5.5: sumbool_of_bool type

```
Definition sumbool_of_bool : ∀ b:bool, {b = true} + {b = false}.
```

With sumbools similar to bool and other 2-constructor inductive types, one can use the “if then else” construct to select the desired case when doing proofs which makes for much more readability of the given proofs.

We have used the sumbool type to declare and prove decision procedures that we have subsequently used in the translation functions implementing the semantics and also in the proofs. We will list the decision procedures in Section 5.4 after reviewing the translation functions in Section 5.3.

5.2 Types of Decisions and their Implementation in Coq

Policy based access-control languages typically use a two-valued decision set to indicate whether an access request is granted or denied. When a decision for a query is not granted, one design choice for a language is to return an explicit deny decision. However in this case deny stands for “not permitted”. It is possible to have cases when the policy truly doesn’t specify either a permit or a deny decision. In such cases arbitrarily returning the decision of deny makes it difficult to compose policies and in fact, an explicit decision of “non applicable” is warranted in such cases. Some languages may decide to only support permit decisions. In such languages lack of a permit decision for a query signifies a deny decision so deny decisions are not explicit. Although the policies of these languages may be more readable than those with more explicit decisions, they result in ambiguity on whether a deny decision was really intended or not. Finally some languages define an explicit decision of “error” for cases such as when both permit and deny decisions are reached for the same query. An explicit error decision is preferable to undefined behaviour because it can lead to improvements to policies and/or how the queries are built [TK06].

Pucella and Weissman [PW06] employ a two-valued decision set where the granting decision is called Permitted and the non-granting or denying decision is called NotPermitted. In ACCPL we have extended the decision set to be three-valued. We have added the Unregulated decision to Permitted and NotPermitted decisions.

The policy translation functions ultimately will return one of the following answers: Permitted, NotPermitted or Unregulated. Note that since ACCPL was inspired by DRM systems and their policy languages (RELs) we will use “unregulated” as synonymous with “non-applicable” for the case where a given policy does not apply to a request.

A Permitted answer signifies that the access request has been granted. The NotPermitted answer is for when the access request is denied. And finally the Unregulated answer is for all the times when neither Permitted or NotPermitted answers are applicable. The answer type in the Coq listing 5.6 implements the answer concept. We wrap answers in a result type and add some context. Intuitively a result will tell us whether a subject may perform an action on an asset or not, or else that the request is unregulated. In listing 5.6 we also show the definition of the helper function makeResult that when given an answer, a subject, an act and an asset builds and returns a result.

Listing 5.6: Decision Procedures: answer and result types

```
Inductive answer : Set :=
  | Permitted : answer
  | Unregulated : answer
  | NotPermitted : answer.

Inductive result : Set :=
  | Result : answer → subject → act → asset → result.

Definition makeResult (ans:answer)(s:subject)(ac:act)(ass:asset) : result :=
 (Result ans s ac ass).
```

5.3 Translations

Intuitively a query or request asks the following question given an agreement: "May subject s perform an action ac to asset a?". We represent a query by its components, namely the subject, action and asset that form the query question: action_from_query, subject_from_query and asset_from_query. While developing the proofs for the correctness of ACCPL we realized we needed this query specific information deep in the translation functions to be able to render unambiguous decisions and ultimately make the proofs work. We therefore pass the query components to all the translation functions starting with the trans_agreement function (see listing 5.9).

In the following paragraphs, we will describe each translation function in detail starting with the trans_agreement function, however a high-level description of how the main algorithm (encoded in these translation functions) works is now in order. We will show the main algorithm in two separate listings based on whether the policy set in question is inclusive or exclusive.

The first listing (Algorithm 30) for inclusive policy sets, shows how a positive answer to a query in the form of a Permitted decision is reached. All cases when a decision of Unregulated is rendered are explicitly captured and shown. The second listing (Algorithm 31) for exclusive policy sets, shows how a negative answer to a query in the form of a NotPermitted decision is reached. This listing also shows that a positive decision of Permitted is reached in exactly the same way as the case for inclusive policy sets. All cases when a decision of Unregulated is rendered are explicitly captured and shown.

Pattern matching in the body of the trans_agreement function (in listing 5.9), on ag, the agreement in question, extracts the components of the agreement. These are ps, the

Algorithm 30 Access Decision Pseudocode: Inclusive Policy Sets

```
if $asset\_from\_query = asset\_from\_agreement$ then
  if $subject\_from\_query$ is in $prin\_u$ then
    if The preRequisite from the policy set holds then
      if The preRequisite from the policy holds then
        if $action\_from\_query = action\_from\_agreement$ then
          $result = subject\_from\_query$ is Permitted to perform $action\_from\_query$
          on $asset\_from\_query$
        else
          $result = Unregulated$
        end if
      else
        $result = Unregulated$
      end if
    else
      $result = Unregulated$
    end if
  else
    $result = Unregulated$
  end if
else
  $result = Unregulated$
end if
```

Algorithm 31 Access Decision Pseudocode: Exclusive Policy Sets

```
if $asset\_from\_query = asset\_from\_agreement$ then
  if $subject\_from\_query$ is in $prin\_u$ then
    if The $preRequisite$ from the policy set holds then
      if The $preRequisite$ from the policy holds then
        if $action\_from\_query = action\_from\_agreement$ then
          $result = subject\_from\_query$ is $Permitted$ to perform $action\_from\_query$
          on $asset\_from\_query$
        else
          $result = Unregulated$
        end if
      else
        $result = Unregulated$
      end if
    else
      $result = Unregulated$
    end if
  else
    if $action\_from\_query = action\_from\_agreement$ then
      $result = subject\_from\_query$ is $NotPermitted$ to perform $action\_from\_query$
      on $asset\_from\_query$
    else
      $result = Unregulated$
    end if
  end if
else
  $result = Unregulated$
end if
```

policy set, prin_u, the agreement's user(s), and a, the asset mentioned in the agreement. Notice that formal argument e of type environment is passed as an argument to many translation functions but will only eventually be used to get the count information from the getCount function. As mentioned above, the components of the query or request, namely action_from_query, subject_from_query and asset_from_query make up the other parameters that are passed into the next level translation function, which is the translation for a policy set called trans_ps (in listing 5.10).

The nonempty list of results returned by the agreement translation function will have a result per primitive policy (primPolicy) in the agreement. As we will see later when we discuss the proofs, we have proven that results containing a Permitted answer and a NotPermitted answer are mutually exclusive. Therefore we interpret the existence of a single (or more) Permitted result in the nonempty list, as the decision being "Permitted" whereas the existence of a single (or more) NotPermitted type result is interpreted as "NotPermitted". We also have a proof that shows that when there exists no Permitted or NotPermitted results in the returned nonempty list, only Unregulated results are possible. In this case we conclude the decision is "Unregulated".

Listing 5.9: Translation of Agreement

```
Definition trans_agreement
  (ag:agreement)(e:environment)(action_from_query:act)
  (subject_from_query:subject)(asset_from_query:asset) : nonemptylist result :=

  match ag with
  | Agreement prin_u a ps ⇒
     (trans_ps e action_from_query subject_from_query asset_from_query ps prin_u
     a)
  end.
```

Translation of a policy set (called trans_ps in listing 5.10), takes as input e, the environment, the subject, action and asset coming from a query: action_from_query, subject_from_query and asset_from_query, ps, the policy set, prin_u, the agreement's user, and a, the asset mentioned in the agreement.

The translation starts with checking that there is a proof for the equality of asset_from_query and a (the asset from the agreement). This is done by using the decision procedure eq_nat_dec (see listing 5.34). If so, the translation function recurses on the composing primPolicySet and calls the local function process_single_ps. Otherwise, the Unregulated answer is wrapped along with the subject, action and asset coming from a query: action_from_query, subject_from_query and asset_from_query by calling the makeResult helper function. Intuitively we are just explicitly stating that policies are not applicable to queries about assets that are not mentioned in the agreement.

The primPolicySet which was passed to process_single_ps was either constructed by a PIPS constructor or by a PEPS constructor, which distinguish between primInclusivePolicySet and primExclusivePolicySet types. The translation functions trans_policy_PIPS (see listing 5.11) and trans_policy_PEPS (see listing 5.12) are called in turn as the case may be.

Listing 5.10: Translation of Policy Set

```
Fixpoint trans_ps
  (e:environment)(action_from_query:act)(subject_from_query:subject)(
     asset_from_query:asset)
  (ps:policySet)
  (prin_u:prin)(a:asset){struct ps} : nonemptylist result :=

let process_single_ps := (fix process_single_ps (pps: primPolicySet):=
  match pps with
    | PIPS pips ⇒
        match pips with
          | PrimitiveInclusivePolicySet prq p ⇒
            (trans_policy_PIPS e prq p subject_from_query prin_u a action_from_query
     )
        end
    | PEPS peps ⇒
        match peps with
          | PrimitiveExclusivePolicySet prq p ⇒
            (trans_policy_PEPS e prq p subject_from_query prin_u a action_from_query
     )
        end
   end) in

if (eq_nat_dec asset_from_query a)
then (∗ asset_from_query = a ∗)
    match ps with
      | PPS pps ⇒ process_single_ps pps
    end
else (∗ asset_from_query <> a ∗)
      (Single
         (makeResult
            Unregulated subject_from_query action_from_query asset_from_query)).
```

The `trans_policy_PIPS` translation function for a `primInclusivePolicySet` checks whether there is a proof for the subject in question, `x`, being in `prin_u`, by calling the decision procedure `trans_prin_dec` (see listing 5.36). If so there is a check for whether the `preRequisite` from the policy set holds or not by calling the decision procedure `trans_preRequisite_dec` (see listing 5.41). If so the translation function `trans_policy_positive` is called. In all other cases `trans_policy_unregulated` (see listing 5.18) is called.

Listing 5.11: Translation of Primitive Inclusive Policy Set

```
Definition trans_policy_PIPS
 (e:environment)(prq: preRequisite)(p:policy)(x:subject)
 (prin_u:prin)(a:asset)(action_from_query:act) : nonemptylist result :=

  if (trans_prin_dec x prin_u)
  then (* prin *)
   if (trans_preRequisite_dec e x prq (getId p) prin_u)
   then (* prin /\ prq *)
    (trans_policy_positive e x p prin_u a action_from_query)
   else (* prin /\ ~prq *)
    (trans_policy_unregulated e x p a action_from_query)
  else (* ~prin *)
   (trans_policy_unregulated e x p a action_from_query).
```

The trans_policy_PEPS translation function for a primExclusivePolicySet checks whether there is a proof for the subject in question, x, being in prin_u, by calling the decision procedure trans_prin_dec. If not, the translation function trans_policy_negative is called. This is because we are translating a primExclusivePolicySet so the policy set is exclusive to the agreement's users implying all subjects, not in the prin_u are denied access. In case there is a proof for the subject x being in prin_u, there is a check for whether the prerequisite from the policy set holds or not (by calling the decision procedure trans_preRequisite_dec). If so, the translation function trans_policy_positive (see listing 5.13) is called, else trans_policy_unregulated is called.

Listing 5.12: Translation of Primitive Exclusive Policy Set

```
Definition trans_policy_PEPS
 (e:environment)(prq: preRequisite)(p:policy)(x:subject)
 (prin_u:prin)(a:asset)(action_from_query:act) : nonemptylist result :=

 if (trans_prin_dec x prin_u)
 then (* prin *)
  if (trans_preRequisite_dec e x prq (getId p) prin_u)
  then (* prin /\ prq *)
   (trans_policy_positive e x p prin_u a action_from_query)
  else (* prin /\ ~prq *)
   (trans_policy_unregulated e x p a action_from_query)
 else (* ~prin *)
  (trans_policy_negative e x p a action_from_query).
```

The trans_policy_positive translation function (see listing 5.13) starts by calling the function trans_pp_list_trans_policy_positive (see listing 5.14) on a list of primPolicys.

Listing 5.13: Translation of Positive Policies

```
Definition trans_policy_positive
 (e:environment)(x:subject)(p:policy)(prin_u:prin)(a:asset)
 (action_from_query: act) : nonemptylist result :=
 match p with
     | Policy pp_list ⇒ trans_pp_list_trans_policy_positive pp_list e x prin_u a
    action_from_query
 end.
```

The function trans_pp_list_trans_policy_positive (see listing 5.14) handles the case of a single primitive policy by calling the handler function process_single_pp_-trans_policy_positive (see listing 5.15). When there is more than a single primitive policy, the function trans_pp_list_trans_policy_positive appends the results of processing a single primitive policy to the results for the rest of the set.

Listing 5.14: Translation of Primitive Positive Policies

```
Fixpoint trans_pp_list_trans_policy_positive
  (pp_list:nonemptylist primPolicy)(e:environment)(x:subject)
  (prin_u:prin)(a:asset)(action_from_query: act){struct pp_list}:=
   match pp_list with
     | Single pp1 ⇒ process_single_pp_trans_policy_positive pp1 e x prin_u a
      action_from_query
     | NewList pp pp_list' ⇒ app_nonempty
    (process_single_pp_trans_policy_positive pp e x prin_u a action_from_query)
    (trans_pp_list_trans_policy_positive pp_list' e x prin_u a action_from_query)
   end.
```

The function process_single_pp_trans_policy_positive acting on a single primPolicy (see listing 5.15), first checks whether the prerequisite from the policy holds or not, by calling the decision procedure trans_preRequisite_dec. If so the translation function checks that there is a proof for the equality of action_from_query and action (the action from the agreement) by calling the decision procedure eq_nat_dec. If so, the translation function finally returns a result of Permitted. In all other cases, a result of Unregulated is returned. Intuitively we are just explicitly stating that policies are not applicable to queries about actions that are not mentioned in the agreement.

Listing 5.15: Translation of a Primitive Positive Policy

```
Definition process_single_pp_trans_policy_positive
   (pp: primPolicy)(e:environment)(x:subject)(prin_u:prin)
   (a:asset)(action_from_query: act)  : nonemptylist result :=
 match pp with
   | PrimitivePolicy prq’ policyId action ⇒
      if (trans_preRequisite_dec e x prq’ (Single policyId) prin_u)
      then (∗ prin /\ prq /\ prq’ ∗)
       if (eq_nat_dec action_from_query action)
       then
         (Single
           (makeResult Permitted x action_from_query a))
       else
         (Single
           (makeResult Unregulated x action_from_query a))
      else (∗ prin /\ prq /\ ~prq’ ∗)
       (Single
           (makeResult Unregulated x action_from_query a))
 end.
```

The `trans_policy_negative` translation function (see listing 5.16) starts by calling the function `trans_pp_list_trans_policy_negative` on a list of `primPolicys`. Note that `trans_pp_list_trans_policy_negative` is not listed here since it follows the exact pattern as `trans_pp_list_trans_policy_positive` (see listing 5.14).

Listing 5.16: Translation of Negative Policies

```
Fixpoint trans_policy_negative
 (e:environment)(x:subject)(p:policy)(a:asset)
 (action_from_query: act){struct p} : nonemptylist result :=
 match p with
     | Policy pp_list ⇒ trans_pp_list_trans_policy_negative pp_list x a
   action_from_query
 end.
```

The function `process_single_pp_trans_policy_negative` acting on a single `primPolicy` (see listing 5.17), first checks that there is a proof for the equality of `action_from_query` and `action` (the `action` from the agreement) by calling the decision procedure `eq_nat_dec`. If so, the translation function returns a result of `NotPermitted`. Otherwise, a result of `Unregulated` is returned. Note that `trans_policy_negative` is called from a context where `trans_prin_dec` is not provable, which means that subject `x`, is not in `prin_u`. Also note that compared to `trans_policy_positive` we don’t check for whether `preRequisite` from the policy holds or not, since `trans_policy_negative` is only called for policy sets of type `primExclusivePolicySet`.

Listing 5.17: Translation of a Primitive Negative Policy

```
Definition process_single_pp_trans_policy_negative
   (pp: primPolicy)(x:subject)
   (a:asset)(action_from_query: act)  : nonemptylist result :=
  match pp with
    | PrimitivePolicy prq' policyId action ⇒
        if (eq_nat_dec action_from_query action)
        then
          (Single
            (makeResult NotPermitted x action_from_query a))
        else
          (Single
            (makeResult Unregulated x action_from_query a))
  end.
```

The trans_policy_unregulated translation function (see listing 5.18) starts by calling the function trans_pp_list_trans_policy_unregulated on a list of primPolicys. Note that trans_pp_list_trans_policy_unregulated is not listed here since it follows the exact pattern as trans_pp_list_trans_policy_positive (see listing 5.14).

Listing 5.18: Translation of Unregulated Policies

```
Fixpoint trans_policy_unregulated
  (e:environment)(x:subject)(p:policy)(a:asset)
  (action_from_query: act){struct p} : nonemptylist result :=
  match p with
     | Policy pp_list ⇒ trans_pp_list_trans_policy_unregulated pp_list x a
    action_from_query
  end.
```

The function process_single_pp_trans_policy_unregulated acting on a single primPolicy (see listing 5.19) simply returns a result of Unregulated.

Listing 5.19: Translation of an Unregulated Policy

```
Definition process_single_pp_trans_policy_unregulated
   (pp: primPolicy)(x:subject)
   (a:asset)(action_from_query: act)  : nonemptylist result :=
  match pp with
    | PrimitivePolicy prq' policyId action ⇒
       (Single (makeResult Unregulated x action_from_query a))
  end.
```

Translation of a prin (called trans_prin in listing 5.20) takes as input, a subject x, and a principal p, and return the proposition "subject x is in principal p". More specifically trans_prin proceeds based on whether the principal p is a single subject or a list of subjects. If p is a single subject s, the proposition x=s is returned. Otherwise the disjunction of the proposition x=s with the proposition "subject x is in principal rest" is returned.

Listing 5.20: Translation of Prin

```
Fixpoint trans_prin
  (x:subject)(p: prin): Prop :=

  match p with
    | Single s ⇒ (x=s)
    | NewList s rest ⇒ ((x=s) \/ trans_prin x rest)
  end.
```

The translation of a `prerequisite` (called `trans_preRequisite` in listing 5.21) takes as input an environment `e`, a subject `x`, a prerequisite `prq` to translate, a set of policy identifiers `IDs` (identifiers of policies implied by the `prq`), the agreement's user(s) `prin_u`, and proceeds by case analysis on the structure of the `prerequisite`. A `prerequisite` can only have the constructor `PreRequisite` however, which in turn causes the translation function `trans_andPrqs` to be called on a nonempty list of `primPreRequisite`s. Finally for each `primPreRequisite`, the translation function `trans_primPreRequisite` proceeds by case analysis on the structure of the primitive prerequisite. A primitive prerequisite is either `TruePrq`, `Constraint` or `NotCons`.

In listing 5.21 the translation for `TruePrq` is the proposition `True`, the translations for `Constraint` and `NotCons` simply call respective translation functions for corresponding types namely `trans_constraint` (see listing 5.22) and `trans_notCons` (see listing 5.23.

Listing 5.21: Translation of Prerequisite

```
Definition trans_primPreRequisite
  (e:environment)(x:subject)(prq:primPreRequisite)(IDs:nonemptylist policyId)
  (prin_u:prin) : Prop :=
   match prq with
    | TruePrq ⇒ True
    | Constraint const ⇒ trans_constraint e x const IDs prin_u
    | NotCons const ⇒ trans_notCons e x const IDs prin_u
   end.

Fixpoint trans_andPrqs
    (e:environment)(x:subject)
    (prqs:nonemptylist primPreRequisite)
    (IDs:nonemptylist policyId)(prin_u:prin) {struct prqs}: Prop :=

   match prqs with
     | Single prq ⇒ trans_primPreRequisite e x prq IDs prin_u
     | NewList prq' rest_prqs ⇒
         (trans_primPreRequisite e x prq' IDs prin_u) /\
         (trans_andPrqs e x rest_prqs IDs prin_u)
   end.

Definition trans_preRequisite
  (e:environment)(x:subject)(prq:preRequisite)(IDs:nonemptylist policyId)(prin_u:
     prin) : Prop :=
  match prq with
    | PreRequisite prqs ⇒ trans_andPrqs e x prqs IDs prin_u
  end.
```

The translation of a constraint (called trans_constraint in listing 5.22) takes as input an environment e, a subject x, a constraint const to translate, a set of policy identifiers IDs (identifiers of policies implied by the parent prerequisite), the agreement's user(s) prin_u and proceeds by case analysis on the structure of the constraint. A constraint is either Principal, Count or a CountByPrin. The translation for Principal calls the translation function trans_prin, for the prn that accompanies the const constraint. The translation for Count and CountByPrin both return the proposition which is the translation function trans_count (see listing 5.24).

Listing 5.22: Translation of Constraint

```
Fixpoint trans_constraint
  (e:environment)(x:subject)(const:constraint)(IDs:nonemptylist policyId)
  (prin_u:prin){struct const} : Prop :=
  match const with
    | Principal prn ⇒ trans_prin x prn

    | Count n ⇒ trans_count e n IDs prin_u

    | CountByPrin prn n ⇒ trans_count e n IDs prn
  end.
```

The translation of a `NotCons` (called `trans_notCons` in listing 5.23) takes as input an environment `e`, a subject `x`, a constraint `const` to translate, a set of policy identifiers `IDs` (identifiers of policies implied by the parent prerequisite), the agreement's user(s) `prin_u` and proceeds to return the negation of the proposition `trans_constraint` (see listing 5.22).

Listing 5.23: Translation of Negation of Constraint

```
Definition trans_notCons
  (e:environment)(x:subject)(const:constraint)(IDs:nonemptylist policyId)(prin_u:prin) :
     Prop :=
  ~ (trans_constraint e x const IDs prin_u).
```

The translation of a `Count` or a `CountByPrin` (called `trans_count` in listing 5.24) takes as input an environment `e`, `n` the total number of times the subjects mentioned in `prin_u` may invoke the the policies identified by `IDs`. A local variable `running_total` has the current count of the number of times subjects in `prin_u` have invoked the policies. Finally the proposition `running_total < n` is returned as the translation for a `Count` or a `CountByPrin`. Functions `process_two_lists` and `trans_count_aux` are defined in listings 5.25 and 5.26 respectively.

Note that the only difference between translations for a `Count` and a `CountByPrin` is the additional `prn` parameter for `CountByPrin` which allows for getting counts for subjects not necessarily the same as `prin_u`, the agreement's user(s). We therefore omit the definition for `CountByPrin`.

Listing 5.24: Translation of Count

```
Fixpoint trans_count
  (e:environment)(n:nat)(IDs:nonemptylist policyId)
  (prin_u:prin) : Prop :=
  let ids_and_subjects := process_two_lists IDs prin_u in
  let running_total := trans_count_aux e ids_and_subjects in
  running_total < n.
```

To implement the translation for a `Count` or a `CountByPrin` we start by calling an auxiliary function `process_two_lists` (see listing 5.25) that effectively returns a new list composed of pairs of members of the first list and the second list (the cross-product of the two input lists). In the case of trans_count, the call is "process_two_lists IDs prin_u" which returns a list of pairs of policyId and subject namely ids_and_subjects. ids_and_subjects is then passed to a locally defined function called trans_count_aux.

Listing 5.25: Count Helper: Cross-Product of Two Lists, process_two_lists

```
Section Process_Lists.
Variable X : Set.
Variable Y : Set.

Fixpoint process_two_lists (l1 : nonemptylist X) (l2 : nonemptylist Y) : nonemptylist (
    Twos X Y) :=
let process_element_list := (fix process_element_list (e1 : X) (l2 : nonemptylist Y) :
    nonemptylist (Twos X Y) :=
 match l2 with
   | Single s ⇒ Single (mkTwos e1 s)
   | NewList s rest ⇒ app_nonempty (Single (mkTwos e1 s)) (process_element_list e1
    rest)
 end) in

 match l1 with
   | Single s ⇒ process_element_list s l2
   | NewList s rest ⇒ app_nonempty (process_element_list s l2) (process_two_lists
    rest l2)
 end.
End Process_Lists.
```

The function trans_count_aux (see listing 5.26) returns the current count for a single pair of policyId and subject (the call to getCount which looks up the environment e and returns the current count per each subject and policyId) and for a list of pairs of policyId and subjects, the addition of get_count (for the first pair) and trans_count_auxs (for the rest of the pairs) is returned.

Listing 5.26: Count Helper: trans_count_aux

```
Fixpoint trans_count_aux (e:environment)(ids_and_subjects : nonemptylist (Twos
    policyId subject)) : nat :=
 match ids_and_subjects with
  | Single pair1 ⇒ getCount e (right pair1) (left pair1)
  | NewList pair1 rest_pairs ⇒
     (getCount e (right pair1)(left pair1)) +
     (trans_count_aux e rest_pairs)
 end.
```

At this point we have covered the semantics of ACCPL in terms of translation functions starting with trans_agreement. In the following listings we will show an example agreement from [PW06] expressed as abstract syntax and as ACCPL. We will also show how the fully built agreement looks like as ACCPL constructs and we will show how a number of queries can be asked and what the results would be and whether the results are as expected.

The statement in listing 5.27 expresses that “the asset TheReport may be printed and displayed by Alice and Bob only and with the restriction that Alice and Bob together may print and display the asset five times".

Listing 5.27: Agreement of Example 2.4

```
agreement
 for Alice and Bob and Charlie
 about The Report
 with and[{Alice, Bob}, {Alice, Bob}<count[5]>] → and[True ⇒_{id1} print, True ⇒_{id2}
    display].
```

The policy statement in 5.27 is expressed in ACCPL using Coq constructs, in listing 5.28.

Listing 5.28: Expressing Agreement of Example 2.4 in ACCPL

```
Definition ps_24_p1:primPolicy :=
  (PrimitivePolicy (makePreRequisite (TruePrq)) id1 Print).

Definition ps_24_p2:primPolicy :=
  (PrimitivePolicy (makePreRequisite (TruePrq)) id2 Display).

Definition ps_24_p:policy :=
  (Policy (NewList ps_24_p1 (Single ps_24_p2))).

Definition ps_24_prq1:primPreRequisite :=
  (Constraint (Principal (NewList Alice (Single Bob)))).

Definition ps_24_prq2:primPreRequisite :=
  (Constraint (CountByPrin (NewList Alice (Single Bob)) 1)).

Definition ps_24_prq:preRequisite :=
  (PreRequisite (NewList ps_24_prq1 (Single ps_24_prq2))).

Definition ps_24:primPolicySet :=
  PIPS (PrimitiveInclusivePolicySet
    ps_24_prq ps_24_p).

Definition A24 := Agreement (NewList Alice (Single Bob)) TheReport (PPS ps_24).
```

The policy statement in listing 5.27 as an ACCPL construct is shown, in listing 5.29.

Listing 5.29: Fully Built Agreement of Example 2.4 in ACCPL

```
Agreement (Alice, [Bob]) TheReport
      (PPS
        (PIPS
          (PrimitiveInclusivePolicySet
            (PreRequisite
              (Constraint (Principal (Alice, [Bob])),
               [Constraint (CountByPrin (Alice, [Bob]) 1)])
            (Policy
              (PrimitivePolicy (PreRequisite [TruePrq]) id1 Print,
               [PrimitivePolicy (PreRequisite [TruePrq]) id2 Display])))))
```

The default environment for the example agreement is shown in listing 5.30. We set all the counts to zero initially.

Listing 5.30: The Default Environment for Example 2.4 in ACCPL

```
Definition e_24 : environment :=
 (ConsEnv (make_count_equality Bob id1 0)
  (ConsEnv (make_count_equality Bob id2 0)
   (ConsEnv (make_count_equality Alice id1 0)
    (SingleEnv (make_count_equality Alice id2 0))))).
```

The query in listing 5.31, asks whether Alice may print the asset TheReport. We expect the action to be allowed based on the first primitive policy. The second primitive policy mentions a different action (e.g. Display) so we expect the request is unregulated.

Listing 5.31: Query: May Alice Print The Report for Example 2.4 in ACCPL

```
Eval compute in (trans_agreement e_24 A24 Print Alice TheReport). =
  Result Permitted Alice Print TheReport, [Result Unregulated Alice Print
    TheReport]
```

The query in listing 5.32, asks whether Charlie may print the asset TheReport. We expect the action to be unregulated because Charlie fails to meet the prerequisites specified for the policy set.

Listing 5.32: Query: May Charlie Print The Report for Example 2.4 in ACCPL

```
Eval compute in (trans_agreement e_24 A24 Print Charlie TheReport). =
  Result Unregulated Charlie Print TheReport, [Result Unregulated Charlie Print
    TheReport]
```

The query in listing 5.33, asks whether Alice may display the asset TheReport. We expect the action to be allowed based on the second primitive policy. The first primitive policy mentions a different action (e.g. Print) so we expect the request is unregulated.

Listing 5.33: Query: May Alice Display The Report for Example 2.4 in ACCPL

```
Eval compute in (trans_agreement e_24 A24 Display Alice TheReport). =
  Result Unregulated Alice Display TheReport, [Result Permitted Alice Display
    TheReport]
```

5.4 Decision Procedures

In the following we will list only the declaration of each decision procedure and the accompanying functions. The individual decision procedures were used as a building block to make the final correctness proofs for ACCPL possible. We will refer to some of these listings later when we discuss the translations and/or proofs using the decision procedures. Please refer to the source code at [Sis16] whenever the usage of a decision procedure in not mentioned in the following (declarations only will be shown for some).

The decision procedure `eq_nat_dec` takes two natural numbers, and declares that either there is a proof for their equality or there exists a proof for their inequality (see listing 5.34).

Listing 5.34: Decision Procedures: `eq_nat_dec` [Tea04]

```
Theorem eq_nat_dec : ∀ n m, {n = m} + {n <> m}.
```

The decision procedure `subject_in_prin_dec` (see listing 5.35) declares that either there is a proof for the proposition `is_subject_in_prin` (see listing 5.35) or there exists a proof for its negation.

Listing 5.35: Decision Procedures: `subject_in_prin_dec`

```
Theorem subject_in_prin_dec :
   ∀ (a:subject) (l:prin), {is_subject_in_prin a l} + {˜ is_subject_in_prin a l}.

Fixpoint is_subject_in_prin (s:subject)(p:prin): Prop :=
  match p with
  | Single s’ ⇒ s=s’
  | NewList s’ rest ⇒ s=s’ \/ (is_subject_in_prin s rest)
  end.
```

The decision procedure `trans_prin_dec` declares that either there is a proof for the proposition `trans_prin` or there exists a proof for its negation (see listing 5.36). The translation function `trans_prin` is the proposition “subject x is in prin” (see listing 5.20).

Listing 5.36: Decision Procedures: `trans_prin_dec`

```
Theorem trans_prin_dec :
   ∀ (x:subject)(p: prin), {trans_prin x p} + {˜trans_prin x p}.
```

The decision procedure `trans_count_dec` declares that either there is a proof for the proposition `trans_count` or there exists a proof for its negation (see listing 5.37). Recall the translation function trans_count is mostly the proposition running_total < n with some local context (see listing 5.24).

Listing 5.37: Decision Procedures: trans_count_dec

```
Theorem trans_count_dec:
  ∀ (e:environment)(n:nat)(IDs:nonemptylist policyId)(prin_u:prin),
    {trans_count e n IDs prin_u} + {~ trans_count e n IDs prin_u}.
```

The decision procedure trans_constraint_dec declares that either there is a proof for the proposition trans_constraint or there exists a proof for its negation (see listing 5.38). The translation function trans_constraint is listed in listing 5.22.

Listing 5.38: Decision Procedures: trans_constraint_dec

```
Theorem trans_constraint_dec :
   ∀ (e:environment)(x:subject)(const:constraint)(IDs:nonemptylist policyId)(
    prin_u:prin),
     {trans_constraint e x const IDs prin_u} + {~trans_constraint e x const IDs
    prin_u}.
```

The decision procedure trans_notCons_dec declares that either there is a proof for the proposition trans_notCons or there exists a proof for its negation (see listing 5.40). The translation function trans_notCons is listed in listing 5.23. The proof for trans_notCons_dec not shown here uses two helper theorems. One is another decision procedure, trans_negation_constraint_dec which declares that either there exists a proof for the negation of trans_constraint or there is a proof for the negation of the negation of trans_constraint. The other helper theorem is double_neg_constraint which basically declares that trans_constraint implies its own double negation. The helpers are listed in 5.39.

Listing 5.39: Decision Procedures: trans_negation_constraint_dec

```
Theorem double_neg_constraint:
   ∀ (e:environment)(x:subject)(const:constraint)(IDs:nonemptylist policyId)(prin_u:
    prin),
     (trans_constraint e x const IDs prin_u) → ~~(trans_constraint e x const IDs
    prin_u).

Theorem trans_negation_constraint_dec :
   ∀ (e:environment)(x:subject)(const:constraint)(IDs:nonemptylist policyId)(prin_u:
    prin),
     {~trans_constraint e x const IDs prin_u} + {~~trans_constraint e x const IDs
    prin_u}.
```

Listing 5.40: Decision Procedures: `trans_notCons_dec`

```
Theorem trans_notCons_dec :
  ∀ (e:environment)(x:subject)(const:constraint)(IDs:nonemptylist policyId)(
   prin_u:prin),
    {trans_notCons e x const IDs prin_u} + {~ trans_notCons e x const IDs prin_u
   }.
```

The decision procedure `trans_preRequisite_dec` declares that either there is a proof for the proposition `trans_preRequisite` or there exists a proof for its negation (see listing 5.41). The translation function `trans_preRequisite` is listed in listing 5.21).

Listing 5.41: Decision Procedures: `trans_preRequisite_dec`

```
Theorem trans_preRequisite_dec :
  ∀ (e:environment)(x:subject)(prq:preRequisite)(IDs:nonemptylist policyId)(prin_u:
   prin),
    {trans_preRequisite e x prq IDs prin_u} + {~ trans_preRequisite e x prq IDs
   prin_u}.
```

Chapter 6

Formal Proofs and Correctness

The tension in designing a policy language is usually between how to make the language expressive enough, such that the design goals for the policy language may be expressed, and how to make the policies verifiable with respect to the stated goals. A central question or goal in policy language design is whether a permission is implied by a set of policy statements. This is referred to as the decidability problem in this thesis. For example, Halpern and Weissman [HW08a] show that in general determining whether a permission is implied by a set of policy statements in XrML is undecidable whereas Pucella and Weissman show that answering such questions is decidable but NP-hard for ODRL [PW06].

We had set out to prove formally that the ACCPL language had certifiably correct semantics. Recall that we specified the semantics of ACCPL using a translation from policy statements together with an access request and an environment containing all relevant facts, to decisions. The goal was to use the translation algorithm to determine when a request for accessing a resource is implied by a given policy, when the request is not implied by the policy and finally when the request is not regulated by the given policy. More explicitly we wanted to show that answering access requests always produced a decision and that the translation algorithm terminated on all input policies with either a decision of `Permitted`, or a decision of`NotPermitted` or a decision of neither `Permitted` nor `NotPermitted`. In this chapter we discuss the correctness proofs and other main results we had set out to accomplish. In doing so we will show that the translation algorithm meets its specified goal (as stated earlier), given a policy.

ACCPL started out as a reduced version of Pucella and Weissman's subset of ODRL [PW06] but based on the direction of the proofs we were developing significant changes had to be made to Pucella and Weissman's language to make the proofs possible. An important lesson learned was that the process of designing a certifiably correct policy language cannot be done in isolation since while certifying correctness for the language derives the design one way, expressiveness requirements of the language usually drive it in the opposite direction. Certifying a policy language correct cannot be a post-design activity as the changes it will impose are often fundamental ones.

In the following section we will list the main theorems that we have proved and the supporting theorems that needed to be proved to lead up to the main results. In general however we will not show the actual proofs in the dissertation (with some exceptions) and instead refer the reader to the source code for ACCPL, including the proofs, which lives at [Sis16].

6.1 Correctness of ACCPL

The theorem `trans_agreement_dec2` is the declaration of the main correctness result for ACCPL (see listing 6.2). Together with other theorems we will describe below, we have "certified" ACCPL correct by proving this theorem. The nonempty list that the agreement translation function `trans_agreement` returns will contain results one per each primitive policy (`primPolicy`) found in the agreement. Specifically the predicate `isResultInQueryResult` takes a `result` and a nonempty list of `result`'s which `trans_agreement` produces, and calls the `In` predicate. The `In` predicate is adapted from the `Coq.Lists.List` standard library and checks for the existence of the input `result` in the nonempty list of results. Definitions for predicates `isResultInQueryResult` and `In` are listed in listing 6.2. The definitions for `answer` and `result` are repeated in the same listing.

Note that by mentioning the agreement translation function directly in the statement of the theorem 6.2, we tie the correcness property to how the translation functions work. To prove the theorem and with each successive subgoal during the interactive proof process, the definition of the translation function in scope gets unfolded and used so the translation functions have to be defined such that each subgoal is discharged and the proof is completed. This is quite different from trying to show a list has some property.

As an example and also a visual aid to understanding how queries are answered, see listing 6.1. The `isResultInQueryResult` predicate looks for a result with an answer of Permitted in the list that `trans_agreement` has produced, for an agreement for three primitive policies (since the set contains three results). In words, we are asking whether Alice is allowed to print the asset ebook, given a policy.

Listing 6.1: Access Request Resulting in Decision of `Permitted`

```
isResultInQueryResult
(Result Permitted Alice Print ebook)
[ (Result Unregulated Alice Print ebook) ; (Result Unregulated Alice Print ebook) ; (
    Result Permitted Alice Print ebook) ]
```

In the case where the whole set is not comprised of `Unregulated` results, we have two mutually exclusive cases. The first case is when the set has a at least one `Permitted` result; we answer the access query in this case with a result of `Permitted` (this would be the case in the listing 6.1). The second case is when the set has at least one `NotPermitted`; we answer the access query in this case with a result of `NotPermitted`.

Listing 6.2: Agreement Translation's Correctness Property

```
Definition isResultInQueryResult (res:result)(results: nonemptylist result) :
    Prop :=
  In res results.

Fixpoint In (a:X) (l:nonemptylist) : Prop :=
   match l with
     | Single s ⇒ s=a
     | NewList s rest ⇒ s = a \/ In a rest
   end.

Inductive answer : Set :=
 | Permitted : answer
 | Unregulated : answer
 | NotPermitted : answer.

Inductive result : Set :=
 | Result : answer → subject → act → asset → result.

Theorem trans_agreement_dec2:
  ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),


 (isResultInQueryResult
   (Result Permitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))
\/

 (isResultInQueryResult
   (Result NotPermitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))
\/

 (~(isResultInQueryResult
   (Result Permitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))
     /\
  ~(isResultInQueryResult
   (Result NotPermitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))
    ).
```

Typically most, if not all of the results will be of type `Unregulated`. In the case where all the results are `Unregulated` we answer the access query with a result of `Unregulated`. We show this case indirectly in the theorem in listing 6.2 by stating the set does not contain a Permitted result nor a NotPermitted result.

6.2 Mutual Exclusivity of Permitted and NotPermitted

The trans_agreement_perm_implies_not_notPerm_dec (see listing 6.3) says that the existence of a Permitted result in the set of results returned by trans_agreement, excludes a NotPermitted result.

Listing 6.3: Permitted result Implies no NotPermitted result

```
Theorem trans_agreement_perm_implies_not_notPerm_dec:
 ∀
   (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

   (isResultInQueryResult
     (Result Permitted subject_from_query action_from_query asset_from_query)
       (trans_agreement e ag action_from_query subject_from_query
    asset_from_query)) →

  ~(isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query asset_from_query)
     (trans_agreement e ag action_from_query subject_from_query
    asset_from_query)).
```

The trans_agreement_NotPerm_implies_not_Perm_dec (see listing 6.4) shows that the existence of a NotPermitted result in the set of results returned by trans_agreement, excludes a Permitted result.

Listing 6.4: NotPermitted result Implies no Permitted result

```
Theorem trans_agreement_NotPerm_implies_not_Perm_dec:
 ∀
   (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

   (isResultInQueryResult
     (Result NotPermitted subject_from_query action_from_query asset_from_query)
       (trans_agreement e ag action_from_query subject_from_query asset_from_query))
     →

  ~(isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
     (trans_agreement e ag action_from_query subject_from_query asset_from_query)).
```

Finally the proof for trans_agreement_not_Perm_and_NotPerm_at_once establishes that both Permitted and NotPermitted results cannot exist in the same set returned by trans_agreement (see listing 6.5). This result also establishes the fact that in ACCPL rendering conflicting decisions is not possible given an agreement.

Note that the "no conflict" result does not apply across agreements. In particular, we don't handle the case of multiple agreements with the same asset and set of users which may be an interesting sub-case. The result applies to a single agreement and specifically to the <policySet> field of an agreement. Recall that the <policySet> element expands into the <policy> element which in turn contains multiple <primPolicy>s each with multiple <preRequisite>s. In this way, we cover the majority of cases that conflict detection algorithms typically cover, including the most relevant ones.

Listing 6.5: Permitted and NotPermitted: Mutually Exclusive

```
Theorem trans_agreement_not_Perm_and_NotPerm_at_once:
  ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),



 ~((isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query asset_from_query))
/\

 (isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query asset_from_query))).
```

The proof for the next theorem trans_agreement_not_NotPerm_and_not_Perm_implies_Unregulated_dec shows that in the case where neither a Permitted nor a NotPermitted result exists in the set returned by trans_agreement, there does exist at least one Unregulated result (see listing 6.6).

Listing 6.6: Not (Permitted and NotPermitted) Implies Unregulated

```
Theorem trans_agreement_not_NotPerm_and_not_Perm_implies_Unregulated_dec:
 ∀
   (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

   (~(isResultInQueryResult
     (Result Permitted subject_from_query action_from_query asset_from_query)
       (trans_agreement e ag action_from_query subject_from_query
    asset_from_query)) /\

   ~(isResultInQueryResult
     (Result NotPermitted subject_from_query action_from_query asset_from_query)
       (trans_agreement e ag action_from_query subject_from_query
    asset_from_query))) →

  (isResultInQueryResult
     (Result Unregulated subject_from_query action_from_query asset_from_query)
      (trans_agreement e ag action_from_query subject_from_query
    asset_from_query)).
```

6.3 Decidable as an Inductive Predicate

We encode the cases discussed in Section 6.1 in an inductively defined predicate called decidable in listing 6.7. Recall that the nonempty list that the agreement translation function trans_agreement returns will return a set of results one per each primPolicy. There are three cases. First when there is at least a NotPermitted result in the set and there is no Permitted result, second when there is at least a Permitted result in the set and no NotPermitted result, and third when there are no Permitted and no NotPermitted results in the set. We encode these three cases in three constructors for the type decidable. The constructors are: Denied, Granted and NotApplicable. Notice that we use Denied, Granted and NotApplicable synonymously with NotPermitted, Permitted and Unregulated. The reason is a practical one and is due to Coq; we needed to use new names for the decidable constructors since we already used NotPermitted, Permitted and Unregulated as constructors for answer.

Listing 6.7: Inductive Predicate decidable

```
Inductive decidable : environment → agreement → act → subject → asset → Prop
     :=

    | Denied : ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

 (isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query asset_from_query)
      (trans_agreement e ag action_from_query subject_from_query
     asset_from_query))
  →
 ~(isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
      (trans_agreement e ag action_from_query subject_from_query
     asset_from_query))

  → decidable e ag action_from_query subject_from_query asset_from_query

    | Granted : ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),
 (isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
      (trans_agreement e ag action_from_query subject_from_query
     asset_from_query))
 →
 ~(isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query asset_from_query)
      (trans_agreement e ag action_from_query subject_from_query
     asset_from_query))

 → decidable e ag action_from_query subject_from_query asset_from_query

    | NonApplicable : ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

  ~(isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query asset_from_query))
     →
  ~(isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query asset_from_query))
  → decidable e ag action_from_query subject_from_query asset_from_query.
```

We declare and prove the correctness of the semantics for ACCPL in terms of the `decidable` predicate as seen in listing 6.8. The proof is made simple since we use the previously proven theorem `trans_agreement_dec2`. The proof also makes use of two theorems we saw earlier in Section 6.2. Recall that the first theorem, `trans_agreement_perm_implies_not_notPerm_dec`, says that the existence of a `Permitted` result in the set of results returned by `trans_agreement`, will exclude any `NotPermitted` result and that the second theorem, `trans_agreement_NotPerm_implies_not_Perm_dec`, shows that the existence of a `NotPermitted` result in the set of results returned by `trans_agreement`, will exclude a `Permitted` result. The proof shows the series of tactics that were applied to complete the proof interactively. We do not explain the tactics here, but just show the proof to illustrate by example what proofs look like.

We will list the intermediate theorems that we have declared and proved which are used in the proof of theorem `trans_agreement_dec2` and subsequent proofs in Section 6.4.

Listing 6.8: Correctness Of Agreement Translation

```
Theorem trans_agreement_decidable:
  ∀
 (e:environment)(ag:agreement)(action_from_query:act)
  (subject_from_query:subject)(asset_from_query:asset),
    decidable e ag action_from_query subject_from_query asset_from_query.

Proof.
intros e ag action_from_query subject_from_query asset_from_query.
specialize trans_agreement_dec2 with
  e ag action_from_query subject_from_query asset_from_query.
intros H. destruct H as [H1 | H2].

apply Granted. assumption.
simpl. intuition.
apply trans_agreement_perm_implies_not_notPerm_dec in H1. contradiction.

destruct H2 as [H21 | H22].
apply Denied. assumption.
simpl. intuition.
apply trans_agreement_NotPerm_implies_not_Perm_dec in H21. contradiction.

apply NonApplicable.
destruct H22 as [H221 H222].
exact H221.
destruct H22 as [H221 H222].
exact H222.
Defined.
```

6.4 Intermediate Theorems

The theorem `trans_agreement_dec_sb_Permitted` states as a `sumbool` that the nonempty list that `trans_agreement` produces, contains a `Permitted` result or not and this is decidable (see listing 6.9).

Listing 6.9: Translation Function for Agreement Returns a `Permitted` result or Not

```
Theorem trans_agreement_dec_sb_Permitted:
 ∀
 (e:environment)(ag:agreement)(action_from_query:act)
  (subject_from_query:subject)(asset_from_query:asset),


{(isResultInQueryResult
   (Result Permitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))} +
{~(isResultInQueryResult
   (Result Permitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))}.
```

The theorem `trans_agreement_dec_sb_NotPermitted` states as a `sumbool` that the nonempty list that `trans_agreement` produces, contains a `NotPermitted` result or not and this is decidable (see listing 6.10).

Listing 6.10: Translation Function for Agreement Returns a `NotPermitted` result or Not

```
Theorem trans_agreement_dec_sb_NotPermitted:
 ∀
 (e:environment)(ag:agreement)(action_from_query:act)
  (subject_from_query:subject)(asset_from_query:asset),

{(isResultInQueryResult
   (Result NotPermitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))} +
{~(isResultInQueryResult
   (Result NotPermitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))}.
```

The theorem `resultInQueryResult_dec` states as a `sumbool` that a given result is either in the nonempty list of `results` or not and this is decidable (see listing 6.11).

Listing 6.11: Result is in Results or Not

```
Theorem resultInQueryResult_dec :
   ∀ (res:result)(results: nonemptylist result),
{isResultInQueryResult res results} + {~isResultInQueryResult res results}.
```

The theorem `trans_agreement_not_Perm_and_NotPerm_at_once` states that you may not have both a `Permitted` and `NotPermitted` result in the nonempty list that `trans_agreement` produces (see listing 6.12).

Listing 6.12: Translation Function for Agreement will not have both `Permitted` and `NotPermitted`

```
Theorem trans_agreement_not_Perm_and_NotPerm_at_once:
  ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),



 ~((isResultInQueryResult
    (Result Permitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query asset_from_query))
/\

 (isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query asset_from_query)
    (trans_agreement e ag action_from_query subject_from_query asset_from_query))).
```

The theorem `trans_policy_PIPS_dec_not` states that the translation function `trans_policy_PIPS` produces a nonempty list of results that will not have a `NotPermitted` result. This makes intuitive sense as `trans_policy_PIPS` is called for inclusive policy sets only (see listing 6.13).

Listing 6.13: Translation Function for PIPS will not return a `NotPermitted` result

```
Theorem trans_policy_PIPS_dec_not:
  ∀
  (e:environment)(prq: preRequisite)(p:policy)(subject_from_query:subject)
  (prin_u:prin)(a:asset)(action_from_query:act),

 ~(isResultInQueryResult
    (Result NotPermitted subject_from_query action_from_query a)
    (trans_policy_PIPS e prq p subject_from_query prin_u a action_from_query)).
```

The theorem `trans_policy_PEPS_perm_implies_not_notPerm_dec` states that having a `Permitted` result in the nonempty list of results that the translation function `trans_policy_PEPS` produces implies there exists no `NotPermitted` result in the nonempty list (see listing 6.14).

Listing 6.14: Permitted implies no NotPermitted

```
Theorem trans_policy_PEPS_perm_implies_not_notPerm_dec:
 ∀
 (e:environment)(prq: preRequisite)(p:policy)(subject_from_query:subject)
 (prin_u:prin)(a:asset)(action_from_query:act),

   (isResultInQueryResult
     (Result Permitted subject_from_query action_from_query a)
       (trans_policy_PEPS e prq p subject_from_query prin_u a action_from_query))
    →

  ~(isResultInQueryResult
     (Result NotPermitted subject_from_query action_from_query a)
      (trans_policy_PEPS e prq p subject_from_query prin_u a action_from_query)).
```

The theorem AnswersNotEqual states that two answers not being equal implies that results built from those answers are not equal as well (see listing 6.15).

Listing 6.15: Non Equal Answers Give Non Equal Results

```
Theorem AnswersNotEqual: ∀ (ans1:answer)(ans2:answer)(s:subject)(ac:act)(ass:asset),
  (ans1<>ans2) → ((Result ans1 s ac ass) <> (Result ans2 s ac ass)).
```

The theorem trans_policy_positive_dec_not states that the translation function trans_policy_positive produces a nonempty list of results that will not have a NotPermitted result (see listing 6.16).

Listing 6.16: Translation Function for Permitting Policies will not return a NotPermitted result

```
Theorem trans_policy_positive_dec_not:
 ∀
(e:environment)(s:subject)(p:policy)(prin_u:prin)(a:asset)
 (action_from_query: act),

  ~(isResultInQueryResult
    (Result NotPermitted s action_from_query a)
    (trans_policy_positive e s p prin_u a action_from_query)).
```

The theorem trans_policy_negative_dec_not states that the translation function trans_policy_negative produces a nonempty list of results that will not have a Permitted result (see listing 6.17).

Listing 6.17: Translation Function for NotPermitting Policies will not return a `Permitted` result

```
Theorem trans_policy_negative_dec_not:
  ∀
(e:environment)(s:subject)(p:policy)(a:asset)
  (action_from_query: act),

  ~(isResultInQueryResult
    (Result Permitted s action_from_query a)
    (trans_policy_negative e s p a action_from_query)).
```

The theorem `trans_policy_unregulated_dec_not` states that the translation function `trans_policy_unregulated` produces a nonempty list of results that will not have a `Permitted` nor a `NotPermitted` result (see listing 6.18).

Listing 6.18: Translation Function for `Unregulated` Requests will not return `Permitted` nor `NotPermitted`

```
Theorem trans_policy_unregulated_dec_not:
  ∀
(e:environment)(s:subject)(p:policy)(a:asset)
  (action_from_query: act),

  ~(isResultInQueryResult
    (Result Permitted s action_from_query a)
    (trans_policy_unregulated e s p a action_from_query)) /\

  ~(isResultInQueryResult
    (Result NotPermitted s action_from_query a)
    (trans_policy_unregulated e s p a action_from_query)).
```

Chapter 7

Expressive Power of ACCPL

We have mentioned previously about the trade-offs we have made in the design of ACCPL between ease of formal proof of correctness and expressive power. We started out by examining Pucella and Weissman's fragment of ODRL [PW06] focusing on what we believed to be the essence or core of their language in terms of expressive power at the same time staying away from some of the more complex constructs in order to achieve our goal of ease of reasoning about the policies written in ACCPL with respect to specific correctness properties we discussed in chapter 1. In the following sections we will review differences between ACCPL and Pucella and Weissman's fragment of ODRL including the syntactic constructs we have left out of ACCPL and discuss whether and how much expressive power has been lost by making ACCPL a small language.

7.1 Requirements

Policies in ODRL as specified by [PW06] are composed of a prerequisite, an action and a unique identifier. A prerequisite is either true, a constraint, a requirement or a condition. Constraints are facts that users don't control such as the constraint that "user must be Bob" (users other than Bob cannot satisfy this constraint). Requirements are facts that users have control over. For example, "Alice has paid 5 dollars" can come true if Alice pays 5 dollars. ODRL also includes constraints that must not hold in the form of conditions.

As mentioned in previous chapters, ACCPL forgoes requirements completely and moves the condition "the negation of a constraint" as a syntactic element under the <constraint> production (see 3.12 and 3.13 in chapter 3). In addition ACCPL excludes the "forEachMember" constraint specified by [PW06]. Requirements in ODRL as specified by [PW06] have a significantly different and more complex translation than other prerequisites because of their dependence on time. For example, the "inSeq[req1, req2]" requirement holds only if "req1" is met before "req2" requirement. The translation for "forEachMember" is a formula that holds if each sub-constraint in "forEachMember" is met by each member mentioned in the constraint. We acknowledge that by removing requirements and the forEachMember constraint, expressive power may have been significantly reduced, however it is not clear how much expressive power is lost when ACCPL is considered as a general purpose access-control language as opposed to a DRM specific policy language. We also anticipate that adding these specific constructs or similar ones to ACCPL or trying to implement them in terms of existing ACCPL constructs would add significant complexity to the semantics and derivation of proofs.

7.2 General Conditions

One reason ACCPL was started small was to establish the correctness proofs for the semantics (as described previously) before we could discuss various ways expressive power could be added to the language. However another reason for the compactness of ACCPL was to keep it generic and suitable for various applications. For example, although we have kept many of the prerequisites from ODRL [PW06], and in general have treated digital rights as our main access-control application throughout the thesis, there is no reason why we cannot add other arbitrary conditions to ACCPL provided they are provable. For example, prerequisites similar in expressive power to XACML targets and/or conditions may be added in ACCPL for a specific application (e.g. access-control in healthcare). Of course, adding new prerequisites would require additional translation functions, decision procedures and potentially new helper lemmas and theorems.

In the following (see listing 7.1), we use abstract syntax to show how an ACCPL policy with XACML style conditions would look like; we have made up a new prerequisite called ’gender’ with possible values ’male’ and ’female’. One would also need to add the gender of users in the environment to enable the translation functions to evaluate whether the gender condition is satisfied or not.

Listing 7.1: ACCPL Policy Using a General Condition

```
agreement
 for Alice and Bob
 about The Report
 with True → and[Alice, count[2], gender[female] $\Rightarrow_{id1}$ print.
```

7.3 PolicySet Combinators

Policy sets in ODRL as specified by [PW06] may be combined using the conjunction operator. This possibility certainly adds to the expressive power of a language, similar to the case of basic policies when they are combined. In ACCPL we have only retained basic policy sets that don’t contain other policy sets. While this means ACCPL is less expressive this design decision was made to make the language respect the “independent composition” property (see section 8.6) and as a consequence be conflict-free (the answer to a access query will be either granting or denying but not both).

Adding the conjunction operator to policy sets in ACCPL should be straight forward in terms of the infrastructure we would needs to add. However, this addition would open the door to possibility of conflicts and additionally the current established results for ACCPL would no longer hold. For conflict detection and reporting, we would add an extra identifier to each policy that signifies which policy set a basic policy belongs to. We would also enhance the `result` data-structure to include both the policy set and policy identifiers in order to help the policy writers fix their policies in case the conflict was not intended. Since allowing policy set conjunction opens the door to conflicting decisions, we can envisage adding policy set combination algorithms such as permit-overrides or deny-overrides to resolve conflicts in the case that the policy writer wouldn't want to fix the policies. Note that conflict detection becomes significantly more difficult if in addition to the combining policy sets, we add requirements that are type dependent (such as ODRL requirements). The conflict detection algorithm would need to take into account overlapping time ranges when defining conflicts. We believe the conflict detection algorithm described by St-Martin and Felty [SF16] could be potentially adapted and used for ACCPL specially given the authors' extensive proof libraries.

In the following we will list some example policies taken from [PW06] that are expressible in ACCPL and some that are not.

7.4 Examples

The statement in listing 7.2 expresses that "the asset TheReport may be printed a total of five times by Alice or Bob" by the first policy statement. The second policy statement is only allowing Alice to print twice more which means Bob is not allowed to use this policy to print at all.

Listing 7.2: Agreement of Example 2.1

```
agreement
 for Alice and Bob
 about The Report
 with True → and[count[5] ⇒_{id1} print, and[Alice, count[2]] ⇒_{id2} print].
```

We show how the policy statement in 7.2 would be expressed in ACCPL using Coq constructs, in listing 7.3.

Listing 7.3: Expressing Agreement of Example 2.1 in ACCPL

```
Definition ps_21_p1:primPolicy :=
  (PrimitivePolicy (makePreRequisite (Constraint (Count  5))) id1 Print).

Definition ps_21_p2prq1:primPreRequisite :=
  (Constraint (Principal (Single Alice))).

Definition ps_21_p2prq2:primPreRequisite :=
  (Constraint (Count 2)).

Definition ps_21_prq:preRequisite :=
  (PreRequisite (NewList ps_21_p2prq1 (Single ps_21_p2prq2))).

Definition ps_21_p2:primPolicy :=
  (PrimitivePolicy ps_21_prq id2 Print).

Definition ps_21_p:policy :=
  (Policy (NewList ps_21_p1 (Single ps_21_p2))).

Definition ps_21:primPolicySet :=
  PIPS (PrimitiveInclusivePolicySet
    (makePreRequisite TruePrq) ps_21_p).

Definition A21 := Agreement (NewList Alice (Single Bob)) TheReport (PPS ps_21).
```

We show how the policy statement in listing 7.2 looks like as an ACCPL construct, in listing 7.4.

Listing 7.4: Fully Built Agreement of Example 2.1 in ACCPL

```
Agreement (Alice, [Bob]) TheReport
         (PPS
            (PIPS
               (PrimitiveInclusivePolicySet (PreRequisite [TruePrq])
                  (Policy
                     (PrimitivePolicy (PreRequisite [Constraint (Count 5)])
                        id1 Print,
                     [PrimitivePolicy
                        (PreRequisite
                           (Constraint (Principal [Alice]),
                            [Constraint (Count 2)])) id2 Print])))))
```

The statement in listing 7.5 expresses that "the asset TheReport may be printed AND displayed a total of five times by Alice or Bob".

Listing 7.5: Agreement of Example 2.2

```
agreement
 for Alice and Bob
 about The Report
 with count[5] → and[True ⇒id1 print, True ⇒id2 display].
```

We show how the policy statement in 7.5 would be expressed in ACCPL using Coq constructs, in listing 7.6.

Listing 7.6: Expressing Agreement of Example 2.2 in ACCPL

```
Definition ps_22_p1:primPolicy :=
  (PrimitivePolicy (makePreRequisite (TruePrq)) id1 Print).

Definition ps_22_p2:primPolicy :=
  (PrimitivePolicy (makePreRequisite (TruePrq)) id2 Display).

Definition ps_22_p:policy :=
  (Policy (NewList ps_22_p1 (Single ps_22_p2))).

Definition ps_22_ps_prq1:primPreRequisite :=
  (Constraint (Count 2)).

Definition ps_22:primPolicySet :=
  PIPS (PrimitiveInclusivePolicySet
    (makePreRequisite ps_22_ps_prq1) ps_22_p).

Definition A22 := Agreement (NewList Alice (Single Bob)) TheReport (PPS ps_22).
```

We show how the policy statement in listing 7.5 looks like as an ACCPL construct, in listing 7.7.

Listing 7.7: Fully Built Agreement of Example 2.2 in ACCPL

```
Agreement (Alice, [Bob]) TheReport
        (PPS
          (PIPS
             (PrimitiveInclusivePolicySet
                (PreRequisite [Constraint (Count 2)])
                (Policy
                  (PrimitivePolicy (PreRequisite [TruePrq]) id1 Print,
                   [PrimitivePolicy (PreRequisite [TruePrq]) id2 Display])))))
```

The statement in listing 7.8 expresses that “the asset TheReport may be printed by Alice 1 time; Bob is not restricted at all so Bob may print the asset any number of times".

Listing 7.8: Agreement of Example 2.3

```
agreement
 for Alice and Bob
 about The Report
 with True → Alice <count[1]> ⇒_{id1} print.
```

We show how the policy statement in 7.8 would be expressed in ACCPL using Coq constructs, in listing 7.9.

Listing 7.9: Expressing Agreement of Example 2.3 in ACCPL

```
Definition ps_23_p1:primPolicy :=
  (PrimitivePolicy
    (makePreRequisite (Constraint (CountByPrin (Single Alice) 1))) id1 Print).

Definition ps_23_p:policy :=
  (Policy (Single ps_23_p1)).

Definition ps_23:primPolicySet :=
  PIPS (PrimitiveInclusivePolicySet
    (makePreRequisite TruePrq) ps_23_p).

Definition A23 := Agreement (NewList Alice (Single Bob)) TheReport (PPS ps_23).
```

We show how the policy statement in listing 7.8 looks like as an ACCPL construct, in listing 7.10.

Listing 7.10: Fully Built Agreement of Example 2.3 in ACCPL

```
Agreement (Alice, [Bob]) TheReport
        (PPS
          (PIPS
            (PrimitiveInclusivePolicySet (PreRequisite [TruePrq])
              (Policy
                [PrimitivePolicy
                  (PreRequisite [Constraint (CountByPrin [Alice] 1)]) id1
                  Print]))))
```

The statement in listing 7.11 expresses that “the asset ebook may be displayed and printed by Alice AND Bob 10 times; under the first policy, Alice and Bob may each display the asset five times, and under the second policy, Alice and Bob may each print the asset 1 time".

Listing 7.11: Agreement of Example 2.5

```
agreement
 for Alice and Bob
 about ebook
 with count[10] → and[forEachMember[{Alice, Bob}; count[5]] ⇒_{id1} display,
                      forEachMember[{Alice, Bob}; count[1]] ⇒_{id2} print].
```

The statement in listing 7.12 expresses that "the asset latestJingle may be played by Alice or Bob, on the condition that an amount of 5$ was paid first and then an attribution to Charlie was made, in that order; the single policy further restricts Alice to only play the asset 10 times". The statement in listing 7.12 is made up of an exclusive policy set, meaning that it restricts actions on assets to the users mentioned explicitly in the agreement, in this case, Alice and Bob.

Listing 7.12: Agreement of Example 2.6

```
agreement
 for Alice and Bob
 about latestJingle
 with inSeq[prePay[5], attribution[Charlie]] ↦ (Alice <count[10]> ⇒_{id1} play).
```

The last two policy statements (listed in 7.11 and in 7.12) are examples of what is not expressible in ACCPL at present. We already covered the missing constructs from ACCPL in section 7.1. In particular for these two examples, we are missing the following constructs: `forEachMember`, `inSeq`, `prePay` and `attribution`.

Chapter 8

ACCPL in the Landscape of Policy-based Policy Languages

Tschantz and Krishnamurthi [TK06] discuss the question of how different policy-based access-control languages may be distinguished from each other. The authors note that although existing studies of complexity and expressive power for access-control policy languages, may ensure tractable verification and the ability to capture certain policies, they do not address classification based on ease of reasoning about policies in a language. Tschantz and Krishnamurthi argue for the need for formal means to compare and contrast policy-based access-control languages with respect to a set of properties common to policy-based access-control languages. They analyze the behaviour of policies in light of additional and/or explicit environmental facts, policy growth and policy decomposition. Tschantz and Krishnamurthi apply their properties to two core policy languages and compare the two languages given the results. One of the core policy languages they present is a simplified XACML and the other is Lithium [HW08b]. In the following sections we will describe Tschantz and Krishnamurthi's properties and evaluate ACCPL with respect to these properties.

8.1 Reasonability Properties

The properties Tschantz and Krishnamurthi [TK06] present revolve around three questions:

1. How decisions change when the environment has more facts
2. Impact of policy growth on how decisions may change
3. How amenable policies are to compositional reasoning

Tschantz and Krishnamurthi [TK06] use a simple policy as a motivating example (which they attribute to [HW08b]) that we will adapt to match our DRM specific application throughout this thesis and reuse here. The example will help to review the range of interpretations possible and ultimately the range of policy language design choices and where ACCPL is on that range. The policy in English is as follows:

- If the subject has a subscription to the Gold bundle, then permit that subject to print `TheReport`.
- If the subject has a subscription to the Basic bundle, then do not permit that subject to print `TheReport`.
- If the subject has no subscription to the Gold bundle, then permit that subject to play Terminator.

Consider a query and a fact in English as: A subscriber to the Basic bundle requests to play the movie Terminator. Let's extract the fact that the subject is a subscriber to the Basic bundle into a separate term and make it part of the environment. The basic policy, the query in question and the environment are all expressed as a (pseudo) logic formula and listed in the listing 8.1. The question is, should the access query be granted?

Listing 8.1: Basic Policy

```
p1 = GoldBundle(s) → Permitted(s, print, TheReport)
p2 = BasicBundle(s) → NotPermitted(s, print, TheReport)
p3 = ~GoldBundle(s) → Permitted(s, play, Terminator)
q1 = (s, play, Terminator)
e1 = BasicBundle(s)
```

According to Tschantz and Krishnamurthi [TK06] at least three different interpretations are possible. We will list the possibilities and then evaluate how ACCPL would answer the query.

- Implicit: grant access because of `p3`. This interpretation assumes that if a subject has a subscription to the Basic bundle, the subject does not have a subscription to the Gold bundle so the assumption is that there is a proof for `¬GoldBundle(s)`. This is an example of "implicit" knowledge and "closed world assumptions".
- Explicit: the policy does not apply to the query. In other words, the query is "NotApplicable". `p1` and `p2` don't apply since the asset and the action in the request don't match the ones in `p1` and `p2`. `p3` does not apply either since no proof exists to show that `¬GoldBundle(s)`.
- Implicit with automatic proof capability: grants access to the query by automatically proving that `¬GoldBundle(s)`. Under this interpretation, proof by contradiction is used to establish `¬GoldBundle(s)`. The assumption `¬GoldBundle(s)` is added to `e1` which already contains `BasicBundle(s)`. Now `p1` permits `TheReport` to be printed by the subject while `p2` would not permit `TheReport` to be printed by the subject, leading to a contradiction and the proof of `¬GoldBundle(s)`.

A policy language based on the first and third interpretations are not certifiable, meaning a machine-checked proof of the correctness of their semantics with respect to different properties may not be possible. The main problem with such languages is the implicitness of their semantics. This was the case for the Pucella and Weissman [PW06] fragment of ODRL; although the interpretation found in Pucella and Weissman's fragment doesn't necessarily match the first and the third interpretations above, they are certainly in the camp of "implicit" interpretations. ACCPL matches the second interpretation in the sense of having explicit semantics which leads to certifiably correct results we discuss elsewhere.

When designing ACCPL and to show the semantics certifiably correct, we sought out all the different combinations and cases possible (through interaction with the proof process) and explicitly enumerated them in the translation functions (see the high-level algorithms in algorithm 30 and in algorithm 31). Once we made explicit all the different cases where a decision was possible, we assigned `Permitted` and `NotPermitted` decisions to two specific cases while every other enumerated case was rendered with an `Unregulated` decision. Explicit enumeration of all the different cases has been certified to be correct and complete by the proof of theorem `trans_agreement_dec2` (in listing 6.2). Note that we could have made different design choices in assigning the decisions.

For example, we could replace all or some of the current `Unregulated` decision assignments to either `Permitted` or `NotPermitted` depending on whether we want to make the language more permissive or more prohibitive. If we want to only use `Permitted` or `NotPermitted`, the current correctness results would no longer hold as is. The `Unregulated` case would have to be removed from the statement of the theorem `trans_agreement_dec2` and the theorem statement to be modified to look like the theorem listed in listing 8.2. A few other intermediate results would also have to be restated and proved, however we fully expect this new theorem and similar ones to be provable in ACCPL.

Listing 8.2: Agreement Translation is Decidable (for `Permitted` or `NotPermitted` only)

```
Theorem trans_agreement_dec:
  ∀
  (e:environment)(ag:agreement)(action_from_query:act)
   (subject_from_query:subject)(asset_from_query:asset),

 (isResultInQueryResult
   (Result Permitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))
\/
 (isResultInQueryResult
   (Result NotPermitted subject_from_query action_from_query asset_from_query)
   (trans_agreement e ag action_from_query subject_from_query asset_from_query))
    .
```

8.2 Policy Combinators

Given that basic policies are often developed by independent entities within an organization, it is natural to want a way of combining them into a higher level policy for the whole organization using policy combinators. Indeed most policy-based access-control languages provide a way to combine policies although in practice, there is a range of possibilities for combining policies.

According to Li et al. [LWQ+09] XACML defines three policy elements: rules, policies, and policy-sets. A rule is the most basic policy element in XACML and is equivalent in concept to ACCPL's primitive policies. In XACML a policy contains rules and a policy set contains policies or other policy sets. Both policies and policy sets provide a way to combine lower level policy elements. A XACML policy specifies a rule-combining algorithm (RCA) while a policy set specifies a policy-combining algorithm (PCA). The RCA specifies how the decisions from the rules are combined to yield one decision while the PCA specifies how the results of evaluating the sub-policies are combined to yield one decision. Some examples of PCAs are: permit-override (if any of the sub policies returns a permit, return only permit), deny-override (if any of the sub policies returns a deny, return only deny) [TK06].

Some languages such as ODRL have nested sub-policies inside other policies where different combinators for different layers may be used to make the language more expressive. According to Pucella and Weissman [PW06] ODRL uses a conjunction operator for both basic policies and policy sets (which contain basic policies).

8.3 Policy Based Access-Control Languages

Tschantz and Krishnamurthi [TK06] formalize the differences between the three interpretations listed above but they also enumerate some common features to all access-control languages. For example, the notion of access requests and decisions are universal in access-control languages. Some languages treat the context or the environment where decisions are rendered separately while other languages may treat environments as part of the access request – regardless, the notion of an environment is a common feature to access-control languages.

Tschantz and Krishnamurthi formally define an access-control policy language as a tuple $L = (P, Q, G, N, \langle\langle . \rangle\rangle)$ where P is a set of policies, Q is a set of requests or queries, G is the granting decisions, N is the non-granting decisions, $\langle\langle . \rangle\rangle$ is a function taking a policy $p \in P$ to a relation between Q and $G \cup N$ where $G \cap N = \emptyset$. Let D represent $G \cup N$. Policy p assigns a decision of $d \in D$ to the query $q \in Q$. L also defines a partial order on decisions such that $d \leq d'$ if either $d, d' \in N$ or $d, d' \in G$ or $d \in N$ and $d' \in G$. Note that for ACCPL, $D = \{\text{Unregulated}, \text{NotPermitted}, \text{Permitted}\}$, $G = \{\text{Permitted}\}$ and $N = \{\text{Unregulated}, \text{NotPermitted}\}$.

8.4 Deterministic and Total

According to Tschantz and Krishnamurthi [TK06] a language L is defined to be deterministic if $forall\ p \in P,\ forall\ q \in Q,\ forall\ d, d' \in D,\ q \langle\langle p \rangle\rangle\, d \wedge q \langle\langle p \rangle\rangle\, d' \implies d = d'$. In words, the definition states that for a deterministic language, given a query and a policy, if the policy translation renders a decision of d and a decision of d', then the two decisions must be the same.

A language L is total if $forall\ p \in P,\ forall\ q \in Q,\ exists\ d \in D$, s.t. $q \langle\langle p \rangle\rangle\, d$. In words, for a total language, the policy translation always renders a decision. As for ACCPL being a total language we believe the fact that ACCPL's semantics have been certified correct (see listing 6.8) implies ACCPL is a total language.

8.5 Safety

As discussed in Section 8.1 policy interpretations could be implicit or explicit. Explicit interpretations distinguish between unknown information and information known to be absent. The explicit approach promotes verbose policies and requests and usually leads to too many `Unregulated` decisions (in the absence of the right environmental facts). However the verbosity could be used to direct what additional facts are needed to make a granting or denying decision. Under the second interpretation in Section 8.1, while the decision is `Unregulated`, the system can direct the entity making the request to provide a proof of ¬`GoldBundle(s)` in order for a granting decision to be rendered.

Implicit interpretations may result in granting unintended access to protected resources because no explicit facts need be mentioned in the environment. Recall that the implicit approach works by making assumptions when facts are not explicitly mentioned. Such unintended assumptions lead to unintended permissions being granted.

Tschantz and Krishnamurthi [TK06] define safety as $forall\ p \in P,\ forall\ q, q' \in Q$, $q \leq q' \implies q \langle\langle p \rangle\rangle\, d \leq q \langle\langle p \rangle\rangle\, d'$ which states that under a safe language L, a request with less (environmental) information, q, will lead to a decision (d) that is less than the decision (d') reached for a request with more (environmental) information such as q' in this definition. Recall that in general, granting decisions are considered greater than non-granting decisions. That is for $d \in N$ and $d' \in G$ we have $d \leq d'$. In particular, in ACCPL we have, $N = \{Unregulated, NotPermitted\}$ and $G = \{Permitted\}$. Policies written in a safe language L, will not result in the leakage of permissions to unintended subjects. If and when incomplete information unintentionally causes an access permission to be granted to an intruder, safety has been violated.

Could we conclude that an access-control policy language with the explicit approach such as ACCPL is also safe? in the design of ACCPL and the translation functions in particular we only grant or deny access in specific cases. We grant access if the asset and the action in query are found in the agreement and `trans_prin` has a proof and `trans_preRequisite` for both policy set and policy have proofs. We deny an access request if the policy set is an exclusive policy set (`primExclusivePolicySet`) and `trans_prin` does not hold. In all other cases we render the decision of `Unregulated`. In other words when all the facts hold, corresponding to the greatest query in the definition of safety, we render a granting decision which is also the greatest decision in the lattice of decisions. In all other cases, where all the facts (proofs) do not hold, we render either a decision of `NotPermitted` or a decision of `Unregulated` which are non-granting decisions, satisfying the definition of safety.

Notice that, technically in ACCPL we have a partially ordered set, a `poset` of decisions and not a lattice. This is because for ACCPL there is no decision acting as the greatest lower bound but we do have a least upper bound in `Permitted`, so we have: `Unregulated` $\leq$ `Permitted` and `NotPermitted` $\leq$ `Permitted`.

8.6 Independent Composition

As discussed in Section 8.1, the third interpretation works by taking into account not only the third policy, $p3$, but also $p1$ and $p2$ to render a decision of granted. Under this interpretation, proof by contradiction was used to establish ¬`GoldBundle(s)`. The assumption ¬`GoldBundle(s)` was added to the assumption `BasicBundle(s)` to make `p1` permit `TheReport` to be printed by the subject while making `p2` not permit `TheReport` to be printed by the subject, leading to a contradiction and hence the proof of ¬`GoldBundle(s)`.

Notice that each of the policies in isolation would result in a `Unregulated` decision under the third interpretation, whereas taking into account all three policies results in a `Permitted` decision. Taking into account all policies and rendering a decision that is not different from combining the decisions reached by each primitive policy in isolation, is a property called "Independent Composition" by Tschantz and Krishnamurthi [TK06]. The third interpretation in Section 8.1 clearly does not have this property. Formally, a language L has the independent composition property, if $forall\ p \in P$, $forall\ q \in Q$, $forall\ d, d^* \in D$, $\boxplus\ (q\,\langle\langle p_1 \rangle\rangle\, d_1,\ q\,\langle\langle p_2 \rangle\rangle\, d_2,\ ...,\ q\,\langle\langle p_n \rangle\rangle\, d_n) = q\,\langle\langle(\oplus p_1, p_2, ..., p_n)\rangle\rangle\, d^*$, where $\oplus$ is a composition operator defined in the language and $\boxplus$ is the decision composition operator. For this definition to be well-defined, there is a requirement that complete environmental facts be available for each individual policy so that a decision can be rendered for each primitive policy with respect to the request, in isolation.

Does ACCPL have the independent composition property? As covered in the chapter on semantics of ACCPL (chapter 5) the translation functions for ACCPL, starting with `trans_agreement` all return sets of `results`. The set of `results` returned by the agreement translation function will have a result per primitive policy in the agreement. Recall that results containing a `Permitted` answer and a `NotPermitted` answer are mutually exclusive and the existence of a single (or more) `Permitted` result in the set makes the whole set "Permitted" whereas the existence of a single (or more) `NotPermitted` result makes the whole set "NotPermitted". When no `Permitted` or `NotPermitted` is seen in the set, we only have `Unregulated` results which would make the whole set "Unregulated". This is the decision combination strategy we have designed for ACCPL; the policy combination strategy ACCPL employs simply preserves all individual decisions and returns the whole set for rendering a global decision. Note that the policy combination strategy employed by ACCPL is what's referred to as "Weak-consensus" strategy according to Li et al. [LWQ$^+$09].

Below we consider the three different possible decisions that the decision combination strategy may reach and review whether the result of any individual policy may change as a result:

1. Global decision is `Permitted`. Only possible results are `Permitted` or `Unregulated`. In this case, only the `Unregulated` results have been modified by the decision combination strategy. `Permitted` results coming from any individual primitive policy will have been preserved.

2. Global decision is `NotPermitted`. Only possible results are `NotPermitted` or `Unregulated`. In this case, similar to the first case, only the `Unregulated` results have been modified by the decision combination strategy. `NotPermitted` results coming from any individual primitive policy will have been preserved.

3. Global decision is `Unregulated`. Only possible results are `Unregulated`. In this case, results coming from any individual primitive policy will have been preserved.

We argue that ACCPL has the independent composition property with respect to granting and denying policies only as detailed in the previous paragraphs.

8.7 Monotonicity of Policy Combinators

Tschantz and Krishnamurthi [TK06] discuss the notion of monotonicity of policy combinators. Intuitively, in a language with a policy combinator that is monotonic, adding another primitive policy, cannot change the combined decision to go from granting to non-granting. In ACCPL, `NotPermitted` results are only possible if `trans_prin` does not hold. Assume the combined (global) decision is already `Permitted` for a certain agreement. In ACCPL, `Permitted` results are only possible if `trans_prin` and the `preRequisite` of the policy set both hold. Adding a primitive policy to the agreement will not change the global `Permitted` result to a `NotPermitted` or an `Unregulated` result, since a primitive policy is composed of a `preRequisite` and an `act` only. There are two cases to consider. Either the `preRequisite` of the added policy holds and the local decision for this policy is another `Permitted` result, or the `preRequisite` of the added policy does not hold, and the local decision for this policy is an `Unregulated` result, which won't affect the combined/global decision of `Permitted`.

Chapter 9

Related Work

Policies in some PBAC languages such as XACML and ODRL are expressed in Extensible Markup Language (XML). Authors of [Yag06] do a survey of XML-based access control languages describing and comparing the following languages and systems: Author-X, FASTER, ODRL, XrML, XML Access Control Language (XACL), Security Assertion Markup Language (SAML), XACML and the language designed by the authors called Semantic Policy Language (SPL). Other examples are WS-Policy, Web Service Description Language (WSDL) and Web Service Policy Language (WSPL) reviewed in [ADdVS04].

Authors of [ADdVS04] argue that XML-based PBAC systems are well suitable for the Internet context where simplicity and easy integration with existing technology and tools must be ensured. XML-based languages are also well suited for the easy interchange of policies [ADdVS04].

While policies in XML-based languages are expressed using XML, the meaning of the policy statements in such languages are expressed using fragments of a natural language like English, resulting in ambiguity with regards to the intended behaviour of the system encoded in these policy statements. The ambiguity in semantics of these languages may lead to implementations varying in their interpretations of the access-control language and ultimately making real the possibility of security breaches (e.g. access granted to unauthorized subjects).

In the following sections we will review related work and approaches to define semantics for PBAC based languages such that one can determine without any ambiguity whether a permission or prohibition follows from a set of policy statements.

9.1 Lithium

Halpern and Weissman [HW08b] use FOL to represent and reason about policies; policies describe the conditions under which a request to perform an action, such as reading a file, is granted or denied. They restrict FOL to get tractability for answering the query of whether a request to access a resource may be granted or denied, given a policy, and argue that despite the tractability results their language is still expressive. They contrast their approach with approaches based on Datalog [CGT89] and point out that Datalog based work is made tractable by restricting function symbols and negation but at the cost of losing some expressive power. For example, Datalog based languages cannot distinguish between rendering explicitly forbidden decisions vs unregulated ones.

Halpern and Weissman [HW08b] focus on satisfying three requirements in the design of Lithium:

> 1. It must be expressive enough to capture in an easy and natural way the policies that people want to discuss.
> 2. It must be tractable enough to allow interesting queries about policies to be answered efficiently.
> 3. It must be usable by non-experts, because we cannot expect policymakers and administrators to be well-versed in logic or programming languages.

In contrast, our main goal was designing a core policy language with certifiably correct semantics such that it could be extended in different ways to add expressiveness as long as the semantics would remain certifiably correct with respect to various results established for ACCPL (e.g. correctness results). However, we note that ACCPL may be a good candidate to satisfy the first goal. We have not (yet) considered the other Lithium design goals.

9.2 Trace-based Semantics

Gunter, et al [GWW01] propose an abstract model and a formal language to express access and usage rights for digital assets. A set of "realities" representing a sequence of *payment* and *render* (e.g. work is rendered by a device) actions make up a license. Semantics in the authors' model are expressed as a function mapping terms of the language to elements of the domain of licenses. The authors argue that their semantics is similar to those used for concurrency where language constructs are modelled as traces of allowed events.

Xiang, et al [XBF08] use Observational Transition System (OTS) modeling to describe licenses to use digital assets. In this formalism licenses and DRM systems are modeled as OTSs, described in CafeOBJ [DF98] which is an algebraic specification language. The authors' approach not only models static properties of licenses, dynamic evolutions or traces of licenses can also be observed and denoted by the actions in OTSs, respectively. Finally, formal verification of licenses expressed in CafeOBJ may be performed using a theorem proving facility which provides an integrated platform for the formal modeling, specification, and verification of licenses and DRM systems. This feature enables one to analyze and prove the licenses as well as DRM systems in a more effective way.

9.3 Semantics Based on Linear Logic

Barth and Mitchell [BM06] express semantics of digital rights using propositional linear logic. Linear logic deals with dynamic properties or finite resources, while classical logic deals with stable truths, or static properties [Gir87]. According to Girard as cited by Lincoln [Lin92], "Linear logic is a resource conscious logic". Bart and Mitchell introduce the notion of monotonicity which captures the idea that acquiring extra rights by a user should not lead to the user having fewer rights and show that the algorithm the Open Mobile Alliance (OMA) uses for assigning actions to rights is non-monotonic. Barth and Mitchell consider whether a sequence of actions complies with a license and show that answering this question for the OMA language is NP-complete. They propose an algorithm based on propositional linear logic to evaluate sequences of actions that is monotonic.

9.4 Automata-based Semantics

Holzer, et al [HKS04] give a semantics for ODRL that models the actions that are allowed according to a contract or an agreement. This model is presented in terms of automata. Each trace through the automaton represents a valid sequence of actions for each participant. The states of the automaton encode the state of the license at each point in time, meaning, which actions are allowed at what point considering the actions that have taken place in the past. ODRL requirements and constraints are modeled as labels that are associated with edges in the automaton: an edge can only be taken if the related requirement is satisfied.

Sheppard and Safavi-Naini [SS09b] point out that Holzer et al, don't present an algorithm in [HKS04] for building their automaton, leading to the conclusion that the examples in their paper are constructed by hand.

9.5 Operational Semantics

Sheppard and Safavi-Naini [SS09b] propose an operational model for both formalizing and enforcing digital rights using a *right expression compiler*. They use their model to develop operational semantics for OMA, from which an interpreter could be derived. The authors argue their semantics provide for

- Actions whose effects are neither instantaneous nor fallible.
- Constraints whose satisfaction changes over time.
- Constraints that modify the form of an action rather than permit or prohibit it outright. The quality and watermark constraints of ODRL, for example, modify an action by altering the resolution of the output and inserting a watermark, respectively.

To show that a PBAC system actually behaves correctly with respect to its specification, proofs are needed, however the proofs that are often presented in the literature (e.g. [HW08b, PW06, TK06]), are hard or impossible to formally verify. The verification difficulty is partly due to the fact that the language used to do the proofs while mathematical in nature, utilizes intuitive justifications to derive the proofs. Intuitive language in proofs means that the proofs could be incomplete and/or contain subtle errors.

9.6 Conflict Detection Algorithms

In the following we will review some related work to ours where the authors have used the Coq Proof Assistant to develop conflict detection algorithms (for policies in particular PBAC languages), to state theorems and specifications about the behaviour of their algorithms, to develop formal proofs of those theorems and finally to machine-check those proofs. The resulting proofs are certified, meaning they provide strong correctness guarantees, due to the fact that they are machine-checked.

Capretta, et al [CSFM07] present a conflict detection algorithm for the Cisco firewall specification [Bon05] and formalize a correctness proof for it in the Coq proof assistant. They define two rules being in conflict as, given an access request, one rule would allow access while the another rule would deny access. The authors present their algorithm in Coq's functional programming language along with access rules and requests which are also encoded in Coq. An OCaml version of the algorithm is extracted and successfully used on actual firewall specifications. The extracted program, according to the authors, can detect conflicts in firewalls with hundreds of thousands of rules. The authors also prove in [CSFM07] that their algorithm finds all conflicts and only the correct conflicts in a set of rules. The algorithm is therefore verified formally to be both sound and complete.

St-Martin and Felty [SF16] represent policies for a fragment of XACML 3.0 in the Coq proof assistant and propose an algorithm for detecting conflicts in XACML policies. They state and prove the correctness of their algorithm in the Coq proof assistant. Their XACML subset includes some complex conditions such as time constraints. The authors compare their work with the conflict detection presented in [CSFM07] and conclude that conflict detection in XACML is more complex and results in having to consider many cases including many subtle corner cases.

St-Martin and Felty's definition of a conflict is the usual one and is similar to the one defined in [CSFM07] for firewall rules: when one rule in a policy permits a request and another denies the same request. Since XACML allows conflicts by design, rule-combining is also provided by XACML. A rule-combining algorithm is meant to resolve conflicts if more than one rule applies. For example, one strategy (or algorithm) for resolving conflicts would be to use the first applicable rule that applies. Policy writers often make use of rule-combining algorithms, but unintended errors such as conflicts are common.

St-Martin and Felty's conflict detection algorithm provides a tool for static analysis and policy debugging and it finds all conflicts, whether intended or not. Policy maintainers may then use the results of running the algorithm on their policies to decide which errors or conflicts were intended and which were not.

Chapter 10

Conclusions and Future Work

10.1 Conclusions

In this thesis, we presented the design and implementation of ACCPL as a small and certifiably correct policy language. ACCPL is a PBAC system that can be used to express general access-control rules and policies. In addition we have defined formal semantics for ACCPL where we have discovered and added all possible cases when answering a query on whether to allow or deny an action to be performed on an asset. We have subsequently used the Coq Proof Assistant to state theorems about the expected behaviour of ACCPL when evaluating a request with respect to a given policy, to develop proofs for those theorems and to machine-check the proofs ensuring correctness guarantees are provided. We have in particular stated, developed and proved a correctness result for the semantics of ACCPL.

We have also studied ACCPL in terms of Tschantz and Krishnamurthi's [TK06] reasonability properties which make policy languages more amenable to reasoning and concluded that, since ACCPL supported these properties, ACCPL was significantly more amenable to analysis and reasoning than other access-control policy languages.

We additionally described why certain design choices were made and how they contributed to the ease of reasoning for ACCPL. Admittedly some expressive power present in other access-control policy languages was omitted from ACCPL in order to achieve the reported correctness proofs. For example, in ACCPL we only support base policy sets (policy sets that are not composed of other policy sets) i.e. no combining of base policy sets using conjunctions or other combining operators are supported in ACCPL.

10.2 Future Work

There are a number of directions that can be taken as future work. In the following we list three distinct directions that could be combined at various stages.

10.2.1 Reasonability Properties

In chapter 8 we presented a set of properties to analyze the behaviour of policies in light of additional and/or explicit environmental facts and policy growth and decomposition [TK06]. We conjectured that ACCPL is deterministic, total, safe, and that it has independent composition property and supports a monotonic policy combinator. However, we have not represented these definitions in Coq for ACCPL, nor have we proved that ACCPL has these properties, as we claim. We defer proving these properties for ACCPL as future work.

10.2.2 Additional Expressivity

A direction for future work is to explore different ways ACCPL could be made more expressive. For example, we can add various policy combinators and their semantics to ACCPL using the translation function framework. The translation function framework we have developed for ACCPL is meant to keep the delicate balance between addition of expressiveness while maintaining provability of current and future results (e.g. current correctness results and future ones mentioned in 10.2.1 respectively).

This future work direction embodies an important lesson we learned while developing ACCPL: the process of designing a certifiably correct policy language cannot be done in isolation since while certifying correctness of the language derives the design one way, expressiveness requirements of the language usually drive it in the opposite direction. Certifying a policy language correct cannot be a post-design activity as the changes it will impose are often fundamental ones.

10.2.3 Certifying other PBAC Systems Correct

A design goal for ACCPL was to make it a target language for deploying policies written in other languages. We could capture, implement and study the semantics of these other policy-based access-control systems using the ACCPL translation function framework and ultimately certify the semantics of those languages with respect to their specifications the same way ACCPL has been certified correct. For example, we can take another PBAC system such as XrML and ODRL, implement them in Coq as additional (or modifications of) existing ACCPL constructs, analyze and reason about them, etc. In the following we outline at a very high-level how one PBAC system in particular can be implemented in Coq using ACCPL.

While XACML is a high-level and platform independent access control system, SELinux [SVLS02] is platform dependent (e.g. Linux based) and low-level. SELinux enhances the Discretionary Access Control (DAC) that most Unix based systems employ by Mandatory Access Control (MAC) where designed access control policies are applied throughout the system possibly overriding whatever DAC is in place by the system users.

SELinux uses Linux's extended file attributes to attach a *security context* to passive entities (e.g. files, directories, sockets) and also to each active entity, typically a Linux user space process. Security context is a data structure that is composed of a user, a role and a domain (or type). While users can map directly to ordinary user names they can also be defined separately. Roles are meant to group users and add flexibility when assigning permissions and are the basis for RBAC in SELinux. Finally domains or types are the basis for defining common access control requirements for both passive and active entities.

The enforcement of SELinux policies are performed by the *security server*. Whenever a security operation is requested from the Linux kernel by a program running in user space, the security server is invoked to arbitrate the operation and either allow the operation or to deny it. Each operation is identified by two pieces of information: an object class (e.g. file) and a permission (e.g. read, write). When an operation is requested to be performed on an object, the class and the permissions associated with the object along with security contexts of the source (typically the source entity is a process) and the object are passed to the security server. The security server consults the loaded policy (loaded at boot time) and allows or denies the access request [SSS04].

The SELinux policy has four different kinds of statements: declarations, rules, constraints and assertions [ALP03]. Assertions are compile time checks that the `checkpolicy` tool performs at compile time. The other three kinds of statements however are evaluated at run-time.

Declaration statements are used to declare types, roles and users. Type declaration statements are used to introduce new types. Roles are declared and authorized for particular domains (types) through role declarations, and finally user declarations are used to define each user and to specify the set of authorized roles for each of these users (see listing 10.1). In the following listings we will present a simplified and modified version of the official SELinux syntax that accommodates better the use of the ACCPL Coq framework.

Listing 10.1: Declaration Statements

```
'type' T ';'                                    ; type T

'role' R T ';'                                  ; role R is associated with type T

'user' U R ';'                                  ; user U is associated with role R
```

Rule statements define access vector rules. Access vector (AV) rules (see listing 10.2) specify which operations are allowed and whether to audit (log). Any operation not covered by AV rules are denied by default and all denied operations are logged. The semantics of the AV rule with `<avkind> allow` is: processes with type `T1` are allowed to perform operations in `P` on objects with class `C` and type `T2`. Note that in our modified syntax for SELinux we will use an explicit `deny` as can be seen in listing 10.2.

Listing 10.2: AV Rule

```
<avRule> ::= <avkind> T1 T2:C P ';'

<avkind> ::= 'allow' | 'deny'
```

When a process changes security context, the role may change, assuming a “role transition” rule exists, relating the old and the new roles. There is a related AV rule called the “type transition” rule where a process with type `T1` is allowed/denied to transition to type `T2` when C=process and P=transition (see listing 10.3).

Listing 10.3: Type Transition and Role-Allow Rules

```
<avkind> T1 T2:process transition ’;’

<role_transition_rule> ::= <avkind> R1 R2 ’;’
    ; when a process changes security contexts this rule must hold
```

Constraints are additional conditions on permissions in the form of boolean expressions that must hold in order for the specified permissions to be allowed (see listing 10.4). Whenever a permission is requested on an object class C, the security server checks that the constraints hold.

Listing 10.4: Constraint Definition

```

<constraint> ::= ’constrain’ C, P, <expr> ’;’

<expr> ::= ’not’ <expr> | <expr> ’and’ <expr> | <expr> or <expr> | U1 <op>
    U2 | T1 <op> T2 | R1 <op> R2

<op> ::= ’==’ | ’!=’
```

We will start by limiting the SELinux policy language to only allow AV rules. As mentioned earlier an operation not covered by an allow rule is denied by default in SELinux proper. We will make up explicit deny rules, such that an agreement is defined to be a combination of allow and deny rules. Allow and deny rules as mappings are defined in listing 10.5.

Listing 10.5: ’allow’/’deny’ Rule as a Mapping

`AV rule :` $T \times (T \times C) \rightarrow 2^P$

Agreements in SELinux are made up of AV rules and other agreements (see listing 10.6).

Listing 10.6: SELinux Agreement

```
<agreement> ::= <avRule> ’;’ <agreement>
```

Environments are collections of *role-type* and *user-role* relations. A role-type relation `role(R, T)` simply associates a role with a type. A user-role relation `user(U, R)` associates a user with a role. An environment is consistent with respect to a security context `<T, R, U>`, if and only if `role(R, T)` and `user(U, R)` relations hold in the environment.

The decision problem in SELinux access control is whether an entity with security context `<T1, R1, U1>` may perform action `P1` to entity with object class `C1` with security context `<T2, R2, U2>`.

To answer such queries we use the authorization relation `auth(C, P, T1, R1, U1, T2, R2, U2)` which is conceptually equivalent to the `Permitted` answer from ACCPL (see listing 10.7).

Listing 10.7: `Permitted` for SELinux

$$\begin{aligned} &\mathit{allow}(T1,T2,C,P) \wedge (E\ \mathit{consistent}\ \mathit{wrt}\ \texttt{<T1, R1, U1>} \wedge \texttt{<T2, R2, U2>}) \wedge \\ &\quad (((C,P) == (\mathit{process},\mathit{transition})) \implies \mathit{allow}(R1,R2)) \\ &\quad \implies \mathit{auth}(C,P,T1,R1,U1,T2,R2,U2) \end{aligned}$$

The negation of the authorization relation `auth(C, P, T1, R1, U1, T2, R2, U2)` is conceptually equivalent to the `NotPermitted` answer from ACCPL (see listing 10.8).

Listing 10.8: `NotPermitted` for SELinux

$$\begin{aligned} &\mathit{deny}(T1,T2,C,P) \vee \neg(E\ \mathit{consistent}\ \mathit{wrt}\ \texttt{<T1, R1, U1>} \wedge \texttt{<T2, R2, U2>}) \vee \\ &\quad (((C,P) == (\mathit{process},\mathit{transition})) \implies \mathit{deny}(R1,R2)) \\ &\quad \implies \neg\mathit{auth}(C,P,T1,R1,U1,T2,R2,U2) \end{aligned}$$

Most policy based access-control languages use a two-valued decision set to indicate whether an access request is granted or denied. The literature for SELinux implies only these two outcomes are possible as well, however formally proving this conjecture in Coq would be an important future work to undertake. Investigating and analyzing various properties when answering queries given an SELinux policy would be another important and interesting future work direction.

References

[ADdVS04] Claudio Agostino Ardagna, Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Xml-based access control languages. *Inf. Sec. Techn. Report*, 9(3):35–46, 2004.

[AJH09] Ehab Al-Shaer, Hongxia Jin, and Gregory L. Heileman, editors. *Proceedings of the 9th ACM Workshop on Digital Rights Management, Chicago, Illinois, USA, November 9, 2009.* ACM, 2009.

[ALP03] Myla Archer, Elizabeth I. Leonard, and Matteo Pradella. Analyzing security-enhanced linux policy specifications. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), 4-6 June 2003, Lake Como, Italy*, page 158. IEEE Computer Society, 2003.

[ANP$^+$03] Anne Anderson, Anthony Nadalin, B Parducci, D Engovatov, H Lockhart, M Kudo, P Humenn, S Godik, S Anderson, S Crocker, et al. eXtensible access control markup language (xacml) version 1.0. *OASIS*, 2003.

[ASLZ08] Masoom Alam, Jean-Pierre Seifert, Qi Li, and Xinwen Zhang. Usage control platformization via trustworthy SELinux. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 245–248. ACM, 2008.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer Science & Business Media, 2004. http://www.labri.fr/perso/casteran/CoqArt/index.html.

[BM06] Adam Barth and John C. Mitchell. Managing digital rights using linear logic. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 127–136. IEEE Computer Society, 2006.

[Bon05] James Boney. *Cisco IOS - in a nutshell: a desktop quick reference for IOS on IP networks (2. ed.).* O'Reilly, 2005.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

[Chl11] Adam Chlipala. *Certified Programming with Dependent Types.* MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

[CPR04] Geoff Collier, Harry Piccariello, and Robby Robson. A digital rights management ecosystem model for the education community. *ECAR (forthcoming), May*, 2004.

[CSFM07] Venanzio Capretta, Bernard Stepien, Amy P. Felty, and Stan Matwin. Formal correctness of conflict detection for firewalls. In Peng Ning, Vijay Atluri, Virgil D. Gligor, and Heiko Mantel, editors, *Proceedings of the 2007 ACM workshop on Formal methods in security engineering, FMSE 2007, Fairfax, VA, USA, November 2, 2007*, pages 22–30. ACM, 2007.

[DF98] Razvan Diaconescu and Kokichi Futatsugi. *Cafeobj Report - The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing.* World Scientific, 1998.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[GWW01] Carl A. Gunter, Stephen Weeks, and Andrew K. Wright. Models and languages for digital rights. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34), January 3-6, 2001, Maui, Hawaii, USA*. IEEE Computer Society, 2001.

[HKS04] Markus Holzer, Stefan Katzenbeisser, and Christian Schallhart. Towards formal semantics for ODRL. In Renato Iannella and Susanne Guth, editors, *Proceedings of the First International Workshop on the Open Digital Rights Language (ODRL), Vienna, Austria, April 22-23, 2004*, pages 137–148, 2004.

[Hue92] Gérard P. Huet. The gallina specification language: A case study. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*, volume 652 of *Lecture Notes in Computer Science*, pages 229–240. Springer, 1992.

[HW08a] Joseph Y. Halpern and Vicky Weissman. A formal foundation for xrml. *J. ACM*, 55(1), 2008.

[HW08b] Joseph Y. Halpern and Vicky Weissman. Using First-Order Logic to Reason about Policies. *ACM Transactions on Information and System Security*, 11(4), 2008.

[Ian02] R. Iannella. Open digital rights language (ODRL) version 1.1. 2002. [Online; accessed 09-February-2015].

[JHM06] Pramod A. Jamkhedkar, Gregory L. Heileman, and Iván Martínez-Ortiz. The problem with rights expression languages. In Moti Yung, Kaoru Kurosawa, and Reihaneh Safavi-Naini, editors, *Proceedings of the Sixth ACM Workshop* *on Digital Rights Management, Alexandria, VA, USA, October 30, 2006*, pages 59–68. ACM, 2006.

[KG10] Andreas Kasten and RÃijdiger Grimm. Making the semantics of ODRL and URM explicit using web ontologies. In *The 8th International Workshop for Technical, Economic and Legal Aspects of Business Models for Virtual Goods*, pages 77–91, Namur, Belgium, 2010.

[Lin92] Patrick Lincoln. Linear logic. *SIGACT News*, 23(2):29–37, May 1992.

[LWQ$^+$09] Ninghui Li, Qihua Wang, Wahbeh H. Qardaji, Elisa Bertino, Prathima Rao, Jorge Lobo, and Dan Lin. Access control policy combining: theory meets practice. In Barbara Carminati and James Joshi, editors, *SACMAT 2009, 14th ACM Symposium on Access Control Models and Technologies, Stresa, Italy, June 3-5, 2009, Proceedings*, pages 135–144. ACM, 2009.

[MPT12] Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Formalisation and implementation of the XACML access control mechanism. In Gilles Barthe, Benjamin Livshits, and Riccardo Scandariato, editors, *Engineering Secure Software and Systems - 4th International Symposium, ESSoS 2012, Eindhoven, The Netherlands, February, 16-17, 2012. Proceedings*, volume 7159 of *Lecture Notes in Computer Science*, pages 60–74. Springer, 2012.

[MRD09] Xavier Maroñas, Eva Rodríguez, and Jaime Delgado. An architecture for the interoperability between rights expression languages based on XACML. In *Proceedings of the 7th International Workshop for technical, economic and legal aspects of business models for virtual goods, Virtual Goods*, 2009.

[NIS09] NIST. A SURVEY OF ACCESS CONTROL MODELS. `http://csrc.nist.gov/news_events/privilege-management-workshop/PvM-Model-Survey-Aug26-2009.pdf`, 2009. [Online; accessed 16-August-2015].

[OAS13] OASIS. *XACML Version 3.0*, 2013. `docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf`.

[PCG$^+$11] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. University of Pennsylvania, 2011. `http://www.cis.upenn.edu/~bcpierce/sf/`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[PRD05] Jose Prados, Eva Rodriguez, and Jaime Delgado. Interoperability between different rights expression languages and protection mechanisms. In *Automated Production of Cross Media Content for Multi-Channel Distribution, 2005. AXMEDIS 2005. First International Conference on*, pages 8–pp. IEEE, 2005.

[PW06] Riccardo Pucella and Vicky Weissman. A formal foundation for ODRL. *CoRR*, abs/cs/0601085, 2006.

[SF16] Michel St-Martin and Amy P. Felty. A verified algorithm for detecting conflicts in XACML access control rules. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 166–175. ACM, 2016.

[Sis16] Bahman Sistany. Source code for ACCPL. `http://www.site.uottawa.ca/~afelty/accpl/`, 2016.

[SS09a] Nicholas Paul Sheppard and Reihaneh Safavi-Naini. On the operational semantics of rights expression languages. In Al-Shaer et al. [AJH09], pages 17–28.

[SS09b] Nicholas Paul Sheppard and Reihaneh Safavi-Naini. On the operational semantics of rights expression languages. In Al-Shaer et al. [AJH09], pages 17–28.

[SSS04] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12. Citeseer, 2004.

[SVLS02] Stephen Smalley, Chris Vance, NAI Labs, and Wayne Salamon. Implementing SELinux as a Linux security module. In *USENIX Technical Conference*, 2002. [Online; accessed 16-February-2015].

[Tea04] The Coq Development Team. *The Coq Proof Assistant Reference Manual.* LogiCal Project, 2004. Version 8.0.

[TK06] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In David F. Ferraiolo and Indrakshi Ray, editors, *SACMAT 2006,11th ACM Symposium on Access Control Models and Technologies, Lake Tahoe, California, USA, June 7-9, 2006, Proceedings*, pages 160–169. ACM, 2006.

[Wik15a] Wikipedia. FairPlay — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/FairPlay`, 2015. [Online; accessed 16-February-2015].

[Wik15b] Wikipedia. Open Web Platform — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/Open_Web_Platform`, 2015. [Online; accessed 16-February-2015].

[Wik15c] Wikipedia. PlayReady — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/PlayReady`, 2015. [Online; accessed 16-February-2015].

[WLD$^+$02] Xin Wang, Guillermo Lao, Thomas DeMartini, Hari Reddy, Mai Nguyen, and Edgar Valenzuela. XrML - eXtensible rights markup language. In Michiharu Kudo, editor, *Proceedings of the 2002 ACM Workshop on XML Security, Fairfax, VA, USA, November 22, 2002*, pages 71–79. ACM, 2002.

[XBF08] Jianwen Xiang, Dines Bjørner, and Kokichi Futatsugi. Formal digital license language with ots/cafeobj method. In *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, pages 652–660. IEEE, 2008.

[Yag06] M Yague. Survey on xml-based policy languages for open environments. *Journal of Information Assurance and Security*, 1(1):11–20, 2006.