Certifying Digital Rights' Expression Languages

by

Bahman Sistany

Ph.D.Thesis Proposal For the Ph.D. degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Ph.D.Thesis Advisor: Amy Felty

Abstract

Digital asset usage rights are expressed and defined using Rights Expression Languages (REL)s. A rule in such languages expresses the access rights of users when accessing a resource. However because of the fact that access rules are expressed using fragments of a natural language like English, it isn't always clear what the intended behavior of the system encoded in the access rules should be. Even when the behavior is formally specified, typically using a subset of predicate logic, proofs that certain properties of the system hold, are hard 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.

In this research, the Coq proof assistant is used to encode and model the behavior of a REL called Open Digital Rights Language (ODRL). Decidablity criteria are expressed and proofs are developed in Coq for a subset of ODRL. Programs will then be extracted from the proofs that can be used directly to render a decision on a sample policy (e.g. whether to allow or deny access to an asset).

The ODRL encodings and the formal behavior model will be adapted and used to investigate whether decidablity results hold for a policy language of a different class, namely Security Enhanced Linux (SELinux) policy language, which is a fine-grained access-control policy language.

Table of Contents

Listings

Chapter 1

Introduction

1.1 Digital Rights

Digital Rights Management (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 and 4) tracking usage.

1.2 Policy Expression Languages

RELs, or more precisely Digital Rights Expression Languages (DREL)s when dealing with digital assets deal with the "rights definition" aspect of the 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.

Currently the most popular RELs are the eXtensible rights Markup Language (XrML) [WLD+02], and the ODRL [Ian02]. Both of these languages are XML based and are considered declarative languages. XrML has been selected 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.

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 *Fair Play* [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.

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 eXtensible Access Control Markup Language (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].

In this thesis we will be generalizing the concept of a policy language from strictly representing subsets of ODRL, to representing (subsets of) both ODRL and SELinux policy languages. We will accomplish this by adapting the ODRL policy language syntax and semantics in Coq to be used for SELinux. The final goal is still performing formal verification on both types of policies, in a unified manner.

1.3 Semantics Of ODRL Policies

Formal methods help ensure that a system behaves correctly with respect to a specification of its desired behavior [Pie02]. This specification of the desired behavior 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 behavior.

To formalize the semantics of ODRL several approaches have been attempted by various authors. Most are logic based [HW08, PW06] while others are based on finite-automata [HKS04], operational semantics based interpreters [SS09] and web ontology (from the Knowledge Representation Field) [KG10]. In this thesis we will focus on the logic based approach to formalizing semantics and will study a specific logic based language that is essentially a subset of ODRL.

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 variables doesn't exist in propositional logic.

A richer logic such as "Predicate Logic", also called "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 [HW08] 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 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 a lot of 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 we will have no way of certifying those programs correct, even assuming the original proofs were in fact correct.

While there are paper-proofs for various properties of ODRL (e.g. the ones in [PW06]), proofs for similar properties do not exist for an important (mandatory) access-control policy system, namely SELinux. In particular, as far as we know, no formal proofs (paper based or otherwise) of decidablity of answering access control queries given an SELinux policy exists in the literature.

1.6 Contributions

In this thesis we will build a language representation framework based on ODRL and definitions in [PW06]. The framework will be formalized in *Coq* [BC04] which is both a programming language and a proof-assistant. We will declare and prove decidablity results of subsets of ODRL all the way up to the complete ODRL fragment defined in [PW06]. We will extract programs from the proofs and demonstrate how they can be used on specific policies to render a specific decision such as "a conflict has been detected".

The reason we start with a specific subset of ODRL (see 2.2) is so that we can concentrate on what believe to be the essence of the language. For example, we will only consider one of three different kinds of "facts" (that may affect the permit/deny type decisions) conjecturing that adding the remaining kinds of facts later will not change the proofs significantly.

Beside "certified decidablity results" for ODRL, we will investigate decidablity for SELinux policies, proving decidablity or show why a proof is not possible (if that is the case) and provide proposals to make the policy language decidable.

Decidablity results will subsume an important sub-category, namely inconsistency or conflict-detection in policy expressions or rules. Authors of [SM12] [SMF13] 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. Authors of [SM12] 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 [SM12].

By using the Coq framework that we are building for ODRL to encode and verify agreements written in a second policy language (and different class of policy language: REL vs access-control) we will demonstrate the suitability of this Coq based framework for other policy languages such as XACML.

This Coq based language framework could also be used to implement and reason about interoperability [PRD05, MRD09] between various policy languages. In this manner our Coq based language (including possible modifications and enhancements due to supporting SELinux) can be viewed as *abstract syntax*, complete with defined semantics, that can be used for implementing various policy languages with more concrete syntax (e.g. W3C's ODRL and SELinux).

1.7 Work Accomplished So Far

The encodings for a subset of ODRL which we call ODRL0 (see 2.2) plus some important functions implementing some of the algorithms in [PW06] have been implemented in Coq. Some of the intermediate theorems have been also been defined and proved.

1.8 Work Remaining

The main decidablity result and its proof for ODRL0 will be completed first. We will add the remaining ODRL constructs incrementally while maintaining decidablity for the main decision algorithm. The remaining constructs include a trouble-some construct ([PW06]), namely $not[policySet]$. We will show this construct does not change the decidablity result already established.

ODRL0 is enough to be used as a basis for SELinux policies without *constrain*s. SELinux constrains are extra conditions that need to be satisfied (in addition to policies) in order for a permission to be granted. We will investigate decidablity for this subset first. We will then add constrains to the ODRL0 subset (as pre-requisites) and investigate decidablity. The study plan calls for August of 2015 for all remaining work to be completed.

Chapter 2

ODRL0 Syntax

2.1 Introduction

Authors of [PW06] use abstract syntax instead of XML (ODRL 2.0 can also be encoded in *JSON* and *RDF/OWL Ontology*) to express statements in the ODRL language. Abstract syntax is a more compact representation than XML in which ODRL policies may be written in and furthermore abstract syntax simplifies specifying the semantics as we shall see later. As an example the agreement "If Mary Smith pays five dollars, then she is allowed to print the eBook 'Treasure Island' twice and she is allowed to display it on her computer as many times as she likes" written in ODRL's XML encoding is illustrated in Listing 2.1 [PW06].

Listing 2.1: Agreement for Mary Smith in XML

```xml
<agreement>
 <asset> <context> <uid> Treasure Island </uid> </context> </
    asset>
 <permission>
   <display>
    <constraint>
     <cpu> <context> <uid> Mary's computer </uid> </context> <
        /cpu>
    </constraint>
   </display>
   <print>
    <constraint> <count> 2 </count> </constraint>
   </print>
  <requirement>
   <prepay>
    <payment> <amount currency="AUD"> 5.00</amount> </payment>
   </prepay>
  </requirement>
 </permission>
```

```
 <party> <context> <name> Mary Smith </name> </context> </
    party>
</agreement>
```

The agreement in Listing 2.1 is shown in 2.2 using the syntax from [PW06].

Listing 2.2: Agreement for Mary Smith as BNF (as used in [PW06])

```
agreement
 for Mary Smith
 about Treasure Island
 with prePay[5.00] -> and[cpu[Mary's Computer] => display,
                                            count[2] => print].
```

In the following we will cover the *abstract syntax* of a subset of ODRL that we later express using Coq's constructs such as *Inductive Types* and Definitions. We will call this subset *ODRL0* both because it is a variation of Pucella's ODRL language and also because it is missing some constructs from Pucella's ODRL (and hence from ODRL proper).

2.2 ODRL0

In ODRL0, agreements and facts (i.e. environments) will only contain the number of times each policy has been used to justify an action. In ODRL0 agreements and facts will not contain:

1. Which payments have been made
2. Which acknowledgments have been made

This means *Paid* and *Attributed* predicates are not used in ODRL0. Also removed are related constructs *prepay* and *attribution*. We also had to remove two other constructs based on *prepay* and *attribution* out of ODRL0 in *inSeq* and *anySeq*. *prepay*, *attribution*, *inSeq* and *anySeq* make up what is called *requirements* in ODRL.

In ODRL a *prerequisite* is either *true*, a *constraint*, a *requirement* or a *condition*. *true* is the prerequisite that always holds. Constraints are facts that are outside of control of users. For example, there is nothing *Alice* can do to satisfy the constraint "user must be Bob". Requirements are facts that are in users' control [PW06]. For example, *Alice* may satisfy the requirement "The user must pay 5 dollars". Finally conditions are constraints that must not hold. The listing 2.2's Coq version is illustrated in listing 2.3. Note that the constructs used in listing 2.3 will be defined later in chapter 3.

Listing 2.3: Coq Version of Agreement for Mary Smith

```
Agreement (Single MarySmith) Treasure Island
 (PrimitivePolicySet (Constraint (PrePay 5.00))
  (AndPolicy
   (NewList (PrimitivePolicy (Constraint
                         (Principal
                          (Single MarysComputer))) id1 Display)
   (Single (PrimitivePolicy (Constraint (Count 2)) id2 Print))))).
```

In ODRL0, a prerequisite is either *true*, a *constraint*, or the negation of a *constraint*. So we have removed requirements from the picture and don't have explicit conditions. Conditions are replaced by a category called *NotCons* directly in the production for prerequisites (see listing 2.12). Note that we have also removed the condition *not*[*policySet*] from ODRL since the authors in [PW06] have shown the semantics of this component are not well-defined and including it leads to intractability results.

We will add the missing pieces as described above making up what we will call *ODRL1* and perhaps *ODRL2* (the latter only if needed). We will also describe ODRL0 in a *BNF* grammar that looks more like Pucella's ODRL grammar [PW06]. BNF style grammars are less formal as they give some 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 more formal and could be directly used for building compilers and interpreters. We will present both the BNF version and the Coq version for each construct of ODRL0.

2.3 Productions

The top level ODRL0 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/users called a *principal* or *prin*, an *asset* and a *policySet*.

Listing 2.4: agreement

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

Principals or prins are composed of *subjects* which are specified based on the application e.g. Alice, Bob, etc.

Listing 2.5: prin

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

Listing 2.6: subject

```
<subject> ::= N
```

Assets are also application specific but similar to subjects we will use specific ones for the DRM application (taken from [PW06]). *ebook*, *The Report* and *latestJingle* are examples of specific subjects we will be using throughout. Syntactically an asset is represented as a natural number (N). Similarly for subjects.

Listing 2.7: asset

```
<asset> ::= N
```

Agreements include policy sets. Each policy set specifies a *prerequisite* and a *policy*. In general if the prerequisite holds the policy is taken into consideration. Otherwise the policy will not be looked at. Some policy sets are specified as *exclusive*. The *Primitive Exclusive Policy Sets* are exclusive to 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 (prin) are forbidden from performing the specified actions. Finally policy sets could be grouped together in a *conjunction* allowing a single agreement to be associated with many policy sets.

Listing 2.8: policySet

```
<policySet> ::=
   <preRequisite> → <policy>              ; primitive policy set
   <preRequisite> ↦ <policy>              ; primitive exclusive policy set
   'and'[ <policySet_1>, ..., <policySet_m> ]
    ; conjunction
```

A policy specifies an action to be performed on an asset, depending of 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. Similar to policy sets, policies could also be grouped together in a conjunction. The policy also includes a unique identifier. The policy identifier is added to help the translation (from agreements to formulas) but is optional in ODRL proper.

Listing 2.9: policy

```
<policy> ::=
   <preRequisite> ⇒_<policyId> <act>       ; primitive policy
   'and'[ <policy_1>, ..., <policy_m> ]     ; conjunction
```

An *Action* (*act*) is represented as a natural number. Similar to assets and subjects, actions are application specific. Some example actions taken from [PW06] are *Display* and *Print*.

Listing 2.10: act

```
<act> ::= N
```

A *Policy Id* (*policyId*) is a unique identifier specified as (increasing) positive integers.

Listing 2.11: policyId

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

In ODRL0 a *prerequisite* 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 user. Minimum height requirements for popular attractions and rides are examples of we would consider a constraint. The constraint *ForEachMember* is interesting in its expressive power but has complicated semantics as we shall see in the 4 chapter. Roughly speaking, ForEachMember takes a prin (a list of subjects) and a list L of constraints. The ForEachConstraint holds if each subject in prin satisfies each constraint in L.*NotCons* is a negation of a constraint. The set of prerequisites are closed under conjunction (*AndPrqs*), disjunction (*OrPrqs*) and exclusive disjunction (*XorPrqs*).

Listing 2.12: preRequisite

```
<preRequisite> ::=
   'True'                                    ; always true
   <constraint>                              ; constraint
   'ForEachMember' [<prin> ; <constraint_1>, ..., <constraint_m> ]
   ; constraint distribution
   'not' [ <constraint> ]                    ; suspending constraint
   'and'[ <preRequisite_1>, ..., <preRequisite_m> ]
   ; conjunction
   'or'[ <preRequisite_1>, ..., <preRequisite_m> ]
   ; disjunction
   'xor'[ <preRequisite_1>, ..., <preRequisite_m> ]
   ; exclusive disjunction
```

Constraints are either *Principal*, *Count* or *CountByPrin*. Principal constraints basically require matching to specified prins. For example, the user being Alice is a Principal constraint. A count constraint refers to a set of policies P and specifies the number of times the user of an agreement has invoked the policies in P to justify her actions. If the count constraint is part of a policy then the set P is composed of the single policy. In the case that the count constraint is part of a policy set, the set P is the set of policies specified in the policy set.

Listing 2.13: constraint

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

Chapter 3

ODRL0 Syntax In Coq

3.1 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 polymorphisn (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 behavior 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].

3.2 ODRL0 Syntax

ODRL0 productions were presented as high level abstract syntax in 2.3. 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 non empty list of *subject*s (see listing 3.1).

Types *asset*, *subject*, *act* and *policyId* 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 $DefinitionAlice$: $subject$:= 101. and the act 'Play' as $DefinitionPlay$: act := 301.. For each "nat" type in ODRL0 we have also used constants that play the role of "Null" objects (see "Null Object Pattern" [MRB98]), for example $NullSubject$. This is needed partly because of the way ODRL0 language elements are defined which corresponds to the need to use *nonemptylist* exclusively even though at intermediate stages during the various algorithms Coq's *list* is a better fit because it allows empty lists.

Next we define the *policySet* datatype. Note the close/one-to-one mapping to its counterpart in listing 2.8. There are three ways a *policySet* can be constructed (see listing 2.8) corresponding to three constructors: *PrimitivePolicySet*, *PrimitiveExclusivePolicySet* and *AndPolicySet*. Both *PrimitivePolicySet* and *PrimitiveExclusivePolicySet* take a *preRequisite* and a *policy* as parameters. Finally *AndPolicySet* takes a non empty list of *policySet*s.

A *policy* is defined as a datatype with constructors *PrimitivePolicy* and *AndPolicy* (see listing 2.9). *PrimitivePolicy* takes a *preRequisite*, a *policyId* and a action *act*. Ignoring the *policyId* for a moment (it is only added to help the translation otherwise *policyId*s don't exist in ODRL proper), a primitive policy consists of a prerequisite and an action. If the prerequisite holds the action is allowed to be performed on the asset. The *AndPolicy* constructor is simply a *nonemptylist* (see listing 3.2) of *policy*s.

The data type *nonemptylist* reflects the definition of "policy conjunction" in listing listing 2.9 in chapter 2 (see definition of *nonemptylist* in listing 3.2). 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 section.

Listing 3.1: 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 :=
 | PrimitivePolicySet : preRequisite → policy → policySet
 | PrimitiveExclusivePolicySet : preRequisite → policy  → policySet
 | AndPolicySet : nonemptylist policySet → policySet.

Inductive policy : Set :=
 | PrimitivePolicy : preRequisite → policyId → act → policy
 | AndPolicy : nonemptylist policy → policy.
```

Listing 3.2: nonemptylist

```

Section nonemptylist.

Variable X : Set.

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

End nonemptylist.
```

In listing 3.3 *preRequisite* is defined as a new datatype with constructors *TruePrq*, *Constraint*, *ForEachMember*, *NotCons*, *AndPrqs*, *OrPrqs* and *XorPrqs* (see listing 2.12 for the abstract syntax equivalent).

TruePrq represents the always true prerequisite. The *Constraint* prerequisite is defined as the type *constraint* so its description is deferred here. 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' if you are 'Bob' (or 'Alice' for that matter). The constructor *ForEachMember* is defined to be a *prin* and a non empty list of *constraints*. Intuitively a *ForEachMember* prerequisite holds if each subject in *prin* satisfies each constraint in the list of *constraints*. 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 5.12). The remaining constructors *AndPrqs*, *OrPrqs* and *XorPrqs* take as parameters non empty lists of prerequisites. They represent conjunction, inclusive disjunction and exclusive disjunction of prerequisites respectively.

Finally a *constraint* (see listing 2.13 for the abstract syntax equivalent) is defined as a new datatype with constructors *Principal*, *Count* and *CountByPrin*.

Principal constraint takes a *prin* to match. For example, the constraint of the user being Bob would be represented as "Principal constraint". The *Count* constructor takes a *nat* which represents the number of times the user of an agreement has invoked the corresponding policies to justify her actions. If the count constraint is part of a policy then the corresponding policies is basically the single policy, whereas in the case that the count constraint is part of a policy set, the corresponding policies would be the set of those policies specified in the policy set. The *CountByPrin* is similar to *Count* but it takes an additional *prin* parameter. In this case the subjects specified in the *prin* parameter override the agreements' user(s).

Listing 3.3: preRequisite

```
Inductive preRequisite : Set :=
 | TruePrq : preRequisite
 | Constraint : constraint → preRequisite
 | ForEachMember : prin  → nonemptylist constraint → preRequisite
 | NotCons : constraint → preRequisite
 | AndPrqs : nonemptylist preRequisite → preRequisite
 | OrPrqs : nonemptylist preRequisite → preRequisite
 | XorPrqs : nonemptylist preRequisite → preRequisite.


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

We started with the encoding of the agreement for Mary Smith in chapter 2/listing 2.3 but we deferred the definition of the Coq constructs used to define that agreement. All the definitions needed to encode the agreement for Mary Smith except for the prerequisite *PrePay* (which is not part of ODRL0) have now been defined. More agreement examples will be given in Chapter 7.

Chapter 4

ODRL0 Semantics

4.1 Introduction

In this section, we describe the semantics of ODRL0 language by a translation from agreements to a subset of many-sorted first-order logic formulas with equality. The semantics will help answer queries of the form "may subject s perform action act to asset a?". If the answer is yes, we say permission is granted. Otherwise permission is denied. Note that in the listings in this chapter we use $[\![]\!]$ (double square brackets) notation as a mapping of ODRL0 syntactic elements to their translations as many-sorted first-order logic formulas. $\triangleq$ are used between a translation and its corresponding formula to mean the translation is "mapped to" the formula. This translation is derived from the one in [PW06] mainly by adapting it to the ODRL0 subset and in particular syntax, which covered in chapter 2.

Unless specifically mentioned, we use superscripts to denote parameters to translation mappings. However we make a distinction on whether the translation notation is used on the right hand side (rhs) of a $\triangleq$ or the left hand side (lhs). The right side occurrence is similar to a function call where we pass actual parameters to the function call - here to the mapping. The left side occurrence is similar to function declarations/definitions where we define formal parameters - here when we define the translation for a construct for the first time.

4.2 Agreement Translation

At a high-level, an agreement is translated into a conjunction of formulas of the form $\forall x(prerequisites(x) \rightarrow P(x))$ where $P(x)$ itself is a conjunction of formulas of the form $prerequisites(x) \rightarrow (\neg)Permitted(x, act, a)$, where $Permitted(x, act, a)$ means the subject x is permitted to perform action act on asset a. More concretely, given an agreement and observing that the only way an agreement can be built is by passing a $prin$ (typically $prin_u$ the agreement's user(s)), an $asset$, and a $policySet$ to the constructor of the type agreement (see 3.2), the given agreement is translated by invoking the agreement translation mapping with the passed in arguments: $[\![agreement]\!]^{policySet, prin_u, a}$.

The agreement translation mapping declares its three formal arguments as a *policySet*, a set of users, *prin*, and an asset, *a* (see 4.1).

Listing 4.1: Agreement Translation

$$[\![agreement]\!]^{policySet,prin,a} \triangleq [\![policySet]\!]^{prin,a}$$

4.3 Policy Set Translation Definition

The translation for a *policySet* declares two formal arguments: a set of users, *prin* and an asset, *a*. The translation is defined by cases, one for each clause of the grammar in listing 2.8. Recall that a *policySet* is either a *PrimitivePolicySet*, a *PrimitiveExclusivePolicySet* or a *AndPolicySet*, a conjunction of policy sets (see 4.2).

Listing 4.2: Policy Set Translation Cases

$$\begin{array}{ll} [\![policySet]\!]^{prin,a} \triangleq & \\ \quad [\![preRequisite \rightarrow policy]\!]^{prin,a} & \texttt{; primitive policy set} \\ \quad [\![preRequisite \mapsto policy]\!]^{prin,a} & \texttt{; primitive exclusive policy set} \\ \quad [\![and[policySet_1, ..., policySet_m]]\!]^{prin,a} & \texttt{; conjunction} \end{array}$$

Each of the *policySet* kinds has its own translation function, which will be defined in the next 3 subsections. See 4.3.4 for the translation of *getId* and 4.3.5 for the translation of *act* mentioned below.

4.3.1 PrimitivePolicySet Translation Definition

Translation of a *PrimitivePolicySet* (*preRequisite* $\rightarrow$ *policy*) declares two formal arguments: a set of users, *prin* and an asset, *a* and yields a formula that includes a test on whether the subject is in the set of agreements' users (translation of *prin*), the translation of the *policy* and the translation of the *preRequisite*. Basically if the subject in question is a user of the agreement and the policySet's prerequisites hold, then the policy holds. Translation of the policy for a *PrimitivePolicySet* is called a *positive translation*. A positive translation is one where the actions described by the policies are permitted (see 4.3).

Listing 4.3: Policy Set Translation Definition : PrimitivePolicySet

$$\begin{array}{l} [\![preRequisite \rightarrow policy]\!]^{prin,a} \triangleq \forall x \ (([\![prin]\!]_x \wedge \\ \quad [\![preRequisite]\!]_x^{getId(policy),prin,a}) \rightarrow [\![policy]\!]_x^{positive,prin,a}) \end{array}$$

4.3.2 PrimitiveExclusivePolicySet Translation Definition

Translation of a *PrimitiveExclusivePolicySet* (*preRequisite* $\mapsto$ *policy*) declares two formal arguments: a set of users, *prin* and an asset, *a* and yields the conjunction of two implications. The first implication, is the same as one found in the translation of $PrimitivePolicySet$. The second implication however restricts access (to make the policy set exclusive) to only those subjects that are in the agreement's user(s). Translation of the policy in the second implication is called a *negative translation*. A negative translation is one where the actions described by the policies are not permitted (see 4.4).

Listing 4.4: Policy Set Translation Definition : PrimitiveExclusivePolicySet

$$[\![preRequisite \mapsto policy]\!]^{prin,a} \triangleq \forall x \ (([\![prin]\!]_x \wedge [\![preRequisite]\!]_x^{getId(policy),prin,a}) \rightarrow [\![policy]\!]_x^{positive,prin,a}) \wedge \forall x \ (\neg[\![prin]\!]_x \rightarrow [\![policy]\!]_x^{negative,prin,a})$$

4.3.3 AndPolicySet Translation Definition

Translation of $AndPolicySet$ declares two formal arguments: a set of users, $prin$ and an asset, a and yields the conjunctions of the corresponding policy set translations (see 4.5).

Listing 4.5: Policy Set Translation Definition : AndPolicySet

$$[\![and[policySet_1, ..., policySet_m]]\!]^{prin,a} \triangleq [\![policySet_1]\!]^{prin,a} \wedge ... \wedge [\![policySet_m]\!]^{prin,a}$$

4.3.4 getId Translation

The $getId$ function when applied to a single *policy*(see 2.9) is defined to return the $policyId$ of the *policy*.

Listing 4.6: getId for a Single policy

$$[\![getId(policy)]\!] \triangleq [\![getId(preRequisite \Rightarrow_{policyId} act)]\!] \triangleq policyId$$

$getId$ function when applied to a set of policies is the set union of the translations for each individual policy(see 4.6).

Listing 4.7: getId for Policies Definition

$$[\![getId(and[policy_1, ..., policy_m)]\!] \triangleq getId(policy_1) \cup ... \cup getId(policy_m)$$

4.3.5 action Translation

Translation of actions, such as *Play* and *Display* is simply the constant corresponding to each action instance.

Listing 4.8: act Translation Definition

$$[\![act]\!] \triangleq N$$

4.4 prin Translation Cases

Translation for a *prin* ($[\![prin]\!]_x$) declares no formal arguments and is a formula that is true if and only if the subject x is in the prin set. A *prin* is either a single subject or a list of subjects so the translation covers both cases (see 4.9).

Listing 4.9: Prin Translation Cases

$$
\begin{aligned}
[\![prin]\!]_x \triangleq \quad & \\
& [\![subject]\!]_x \\
& [\![\{subject_1, ..., subject_m\}]\!]_x
\end{aligned}
$$

Each of the *prin* kinds has its own translation function, which will be defined in the following 2 subsections.

4.4.1 Single Subject Translation Definition

If the *prin* is a single subject, the translation is a formula that is true if and only if the subject x is the same as the single subject *subject* (see 4.10).

Listing 4.10: Prin Translation Definition : Single subject

$$[\![subject]\!]_x \triangleq x = subject$$

4.4.2 List of Subjects Translation Definition

Translation of a list of subjects is the disjunction of the translations for each subject (see 4.11).

Listing 4.11: Prin Translation Definition : List of subjects

$$[\![\{subject_1, ..., subject_m\}]\!]_x \triangleq [\![subject_1]\!]_x \lor ... \lor [\![subject_m]\!]_x$$

4.5 Policy Translation Cases

The translation for a *policy* declares three formal arguments: *polarity*, which indicates whether the policy translation is positive or negative, a set of users, *prin* and an asset, a. The translation is defined by cases, one for each clause of the grammar in listing 2.9. Recall that a *policy* is either a $PrimitivePolicy$ or a $AndPolicy$, a conjunction of policies (see 4.12).

Listing 4.12: Policy Translation Cases

$$\llbracket policy \rrbracket^{polarity,prin,a} \triangleq$$

$\llbracket preRequisite \Rightarrow_{policyId} act \rrbracket_x^{polarity,prin,a}$ `; primitive policy`

$\llbracket and[policy_1, ..., policy_m] \rrbracket_x^{polarity,prin,a}$ `; conjunction`

Each of the *policy* kinds has its own translation function, which will be defined in the next 2 subsections.

4.5.1 PrimitivePolicy Translation Definition

Translation of a $PrimitivePolicy$ ($preRequisite \Rightarrow_{policyId} act$) declares three formal arguments: *polarity*, which indicates whether the policy translation is positive or negative, a set of users, *prin* and an asset, a. In the case of positive polarity, the translation yields a formula that 'permits' x to *act* on a if translation for the *preRequisite* holds. In the case that the polarity is negative, the translation yields a formula indicating that x is not permitted to *act* on a regardless of whether the translation for *preRequisite* holds or not (see 4.13).

Listing 4.13: PrimitivePolicy Translation Definition

$$\llbracket preRequisite \Rightarrow_{policyId} act \rrbracket_x^{polarity,prin,a} \triangleq$$

$(\llbracket preRequisite \rrbracket_x^{\{policyId\},prin,a}) \Rightarrow Permitted(x, \llbracket act \rrbracket, a)$

`; polarity=positive`

$\neg(Permitted(x, \llbracket act \rrbracket, a))$ `; polarity=negative`

4.5.2 AndPolicy Translation Definition

Translation of *AndPolicy* declares three formal arguments: *polarity*, which indicates whether the policy translation is positive or negative, a set of users, *prin* and an asset, a and yields the conjunctions of the corresponding policy translations (see 4.14).

Listing 4.14: Policy Translation Definition : AndPolicy

$$\llbracket and[policy_1, ..., policy_m] \rrbracket^{polarity,prin,a} \triangleq \llbracket policy_1 \rrbracket^{polarity,prin,a} \wedge ... \wedge \llbracket policy_m \rrbracket^{polarity,prin,a}$$

4.6 Prerequisite Translation Cases

Translation for a *preRequisite* declares three formal arguments: a set of *policyId*s (identifiers for policies that are implied by the prerequisites), a *prin* and an asset, a. The translation is defined by cases, one for each clause of the grammar in listing 2.12. Recall that a *preRequisite* is either always $TruePrq$, a $Constraint$, a $ForEachMember$, a $NotCons$, a $AndPrqs$, a $OrPrqs$ or a $XorPrqs$ (see 4.15).

Listing 4.15: Prerequisite Translation Cases

$[\![preRequisite]\!]_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq$
$[\![true]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$ `; always true`
$[\![constraint]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$ `; constraint`
$[\![forEachMember[prin';\{constraint_1,...,constraint_n\}]]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$
`; constraint distribution`
$[\![not\ \ constraint]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$ `; suspending constraint`
$[\![and\ \ [preRequisite_1,...,preRequisite_k]]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$
`; conjunction`
$[\![or\ \ [preRequisite_1,...,preRequisite_k]]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$
`; disjunction`
$[\![Xor\ \ [preRequisite_1,...,preRequisite_k]]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$
`; exclusive disjunction`

Each of the *preRequisite* kinds has its own translation function, which will be defined in the following subsections.

4.6.1 True Prerequisite Translation Definition

The translation for a $TruePrq$ yields a formula that is always *true* (see 4.16).

Listing 4.16: Prerequisite Translation Definition : Always True Prerequisite

$[\![true]\!]_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq$ `True`

4.6.2 Constraint Prerequisite Translation Definition

The translation for a *constraint* declares three formal arguments: a set of *policyIds*, a set of users, *prin* and an asset, *a*. The translation is defined by cases, one for each clause of the grammar in listing 2.13. Recall that a *constraint* is either a $Principal$, a $Count$ or a $CountByPrin$ (see 4.17).

Listing 4.17: Prerequisite Translation Definition : Constraint

$[\![constraint]\!]_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq$
$[\![prin]\!]_x$ `; principal`
$[\![count[N]]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$ `; number of executions`
$[\![(count[N])^{prin'}]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}$ `; number of executions by` $prin'$

Each of the *constraint* kinds has its own translation function, which will be defined in the subsections following 4.7.

4.6.3 ForEachMember Prerequisite Translation Definition

The translation for a $forEachMember$ declares three formal arguments: a set of $policyId$s, a set of users, $prin$ and an asset, a. Note that $forEachMember$ takes additional formal arguments: a $prin'$ that overrides the $prin$ that is passed at the $preRequisite$ level and a set of $constraint$s (see 4.18).

Listing 4.18: Prerequisite Translation Definition : ForEachMember

$$
\begin{aligned}
&[\![forEachMember[prin'; \{constraint_1, ..., constraint_n\}]]\!]_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq \\
&\quad \bigwedge_{(subject,i)\in prin' \times \{1,...,n\})} [\![constraint_i]\!]_x^{\{policyId_1,...,policyId_m\},\{subject\},a}
\end{aligned}
$$

4.6.4 NotCons Prerequisite Translation Definition

The translation for a $NotCons$ yields a formula that is simply the negation of the translation for a constraint (see 4.17).

Listing 4.19: Prerequisite Translation Definition : Not Constraint

$$
[\![not\ \ constraint]\!]_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq \neg[\![constraint]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}
$$

4.6.5 AndPrqs Prerequisite Translation Definition

The translation for a $AndPrqs$ yields a formula that is the conjunction of the translation for each $preRequisite$ (see 4.20).

Listing 4.20: Prerequisite Translation Definition : Conjunction

$$
\begin{aligned}
&[\![and\ \ [preRequisite_1, ..., preRequisite_k]]\!]_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq \\
&\quad [\![preRequisite_1]\!]_x^{\{policyId_1,...,policyId_m\},prin,a} \wedge \ ... \ \wedge \\
&\quad [\![preRequisite_k]\!]_x^{\{policyId_1,...,policyId_m\},prin,a}
\end{aligned}
$$

4.6.6 OrPrqs Prerequisite Translation Definition

The translation for a $OrPrqs$ yields a formula that is the inclusive disjunction of the translation for each $preRequisite$ (see 4.21).

Listing 4.21: Prerequisite Translation Definition : Inclusive Disjunction

$$\llbracket or\ \ [preRequisite_1, ..., preRequisite_k]\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq$$
$$\llbracket preRequisite_1\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a} \vee \ ... \ \vee$$
$$\llbracket preRequisite_k\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a}$$

4.6.7 XorPrqs Prerequisite Translation Definition

The translation for a *XorPrqs* yields a formula that is the exclusive disjunction of the translation for each *preRequisite* (see 4.22).

Listing 4.22: Prerequisite Translation Definition : Exclusive Disjunction

$$\llbracket Xor\ \ [preRequisite_1, ..., preRequisite_k]\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq$$
$$\llbracket preRequisite_1\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a} \oplus \ ... \ \oplus$$
$$\llbracket preRequisite_k\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a}$$

4.7 Constraint Translation Cases

Translation for a constraint is a formula $\llbracket constraint\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a}$, where the set of *policyIds* are identifiers for policies that are implied by the constraint, *prin* is the set of users (typically the agreement's user(s) denoted by $prin_u$) to which the constraint applies, x is a variable of type *subject*.

4.7.1 Principal Constraint Translation

The translation for a *Principal* is defined to be the translation of the corresponding *prin* defined earlier in listing 4.9.

4.7.2 Count Constraint Translation

The translation for a *Count* declares three formal arguments: a set of *policyIds*, a set of users, *prin* and an asset, a and it is a formula that is true when the sum of the number of times that $subject_i$ (taken from *prin*) has invoked a policy with policy identifier id_j is smaller than N (see listing 4.23).

Listing 4.23: Constraint Translation : Count

$$\llbracket count[N]\rrbracket_x^{\{policyId_1,...,policyId_m\},prin,a} \triangleq (\textstyle\sum_{(id,subject)\in(\{policyId_1,...,policyId_m\}\times prin)}$$
$$getCount(subject, id)) < N$$

4.7.3 CountByPrin Constraint Translation

The translation for a $CountByPrin$ declares the same formal arguments as the translation for $Count$ and it is a formula that is true when the sum of the number of times that $subject_i$ has invoked a policy with policy identifier id_j is smaller than N. The difference from the $Count$ case, is that $prin'$ in $CountByPrin$ overrides the passed in $prin$ (typically agreement's user(s) or $prin_u$). See listing 4.24.

Listing 4.24: Constraint Translation : Count by Principal

$$[\![(count[N])^{prin'}]\!]_x^{\{policyId_1,\ldots,policyId_m\},prin,a} \triangleq (\textstyle\sum_{(id,subject)\in(\{policyId_1,\ldots,policyId_m\}\times prin')} getCount(subject, id)) < N$$

Chapter 5

ODRL0 Semantics In Coq

5.1 Introduction

The translation functions plus the auxiliary types and infrastructure which implement the semantics have been encoded in Coq. Translation functions build Coq terms of type *Prop*. Well-formed propositions (or *Prop*s) are assertions one can express about values such as mathematical objects or even programs (e.g. $3 < 8$) in Coq.

Whether a permission is granted or denied depends on the agreements in question but also on the facts recorded in the environment. For ODRL0 those facts revolve around the number of times a policy has been used to justify an action (see section 2.2 for more details on odrl0). We encode this information in an *environment* which is a conjunction of equalities of the form $count(s, policyId) = n$.

The Coq version of the count equality is a new inductive type called *count_equality*. An environment is defined to be a non-empty 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*s. For an example of how environments are created see listing 5.2.

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.
```

Listing 5.2: Defining Environments

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

We also define a *getCount* function (see listing 5.3) that given a pair consisting of a subject and policy id, looks for a corresponding count in the environment. Note the use of the keyword *Fixpoint*. Fixpoint is the Coq keyword for recursive functions and we use it instead of 'Definition' when the function being defined is recursive such as is the case for function *getCount*. *getCount* assumes the given environment is consistent, 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.

Listing 5.3: getCount Function

```
Fixpoint getCount
 (e:environment)(s:subject)(id: policyId): nat :=
 match e with
 | SingleEnv f  =>
    match f with
    | CountEquality s1 id1 n1 =>
       if (beq_nat s s1)
       then if (beq_nat id id1) then n1 else 0
       else 0
    end
 | ConsEnv f rest =>
    match f with
    | CountEquality s1 id1 n1 =>
       if (beq_nat s s1)
       then if (beq_nat id id1) then n1 else (getCount rest s id)
       else (getCount rest s id)
    end
 end.
```

5.2 Translations

Translation of the top level *agreement* element proceeds by case analysis on the structure of the agreement. However an agreement can only be built one way; by calling the constructor *Agreement*. The translation proceeds by calling the translation function for the corresponding *policySet* namely the parameter to *Agreement* called *ps*. See listing 5.4 which is the Coq implementation of 4.1. The formal argument *e* of type *environment* is passed along many translation function but will only eventually be used to get the count information from the *getCount* function.

Listing 5.4: Translation of Agreement

```
Definition trans_agreement (e:environment)(ag:agreement) : Prop :=
 match ag with
  | Agreement prin_u a ps ⇒ trans_ps e ps prin_u a
 end.
```

Translation of a *policySet* (called *trans_ps* in listing 5.5), takes as input e, the environment, ps, the policy set, $prin_u$, the agreement's user, and a, the asset, and proceeds by case analysis of different policySet constructors and recursing into translation functions for the composing elements. A policySet is either a $PrimitivePolicySet$, $PrimitiveExclusivePolicySet$ or a $AndPolicySet$.

Note that to implement the translation for an $AndPolicySet$ a local function $trans_ps_list$ has been defined where for a single *policySet*, $trans_ps$ is called, and for a list of *policySet*s, the conjunction of $trans_ps$s are returned.

The listing 5.5 is the Coq implementation of 4.2.

Listing 5.5: Translation of Policy Set

```
Fixpoint trans_ps
 (e:environment)(ps:policySet)(prin_u:prin)(a:asset){struct ps} : Prop :=

let trans_ps_list := (fix trans_ps_list (ps_list:nonemptylist policySet)(prin_u:prin)
   (a:asset){struct ps_list}:=
 match ps_list with
  | Single ps1 ⇒ trans_ps e ps1 prin_u a
  | NewList ps ps_list' ⇒ ((trans_ps e ps prin_u a) /\ (trans_ps_list ps_list' prin_u a
   ))
 end) in
  match ps with
  | PrimitivePolicySet prq p ⇒ ∀ x, (((trans_prin x prin_u) /\
                     (trans_preRequisite e x prq (getId p) prin_u)) →
                     (trans_policy_positive e x p prin_u a))

  | PrimitiveExclusivePolicySet prq p ⇒ ∀ x, ((((trans_prin x prin_u) /\
                          (trans_preRequisite e x prq (getId p) prin_u)) →
                         (trans_policy_positive e x p prin_u a)) /\
                        ((not (trans_prin x prin_u)) → (trans_policy_negative e
    x p a)))

  | AndPolicySet ps_list ⇒ trans_ps_list ps_list prin_u a
  end.
```

Translation of a *prin* (called *trans_prin* in listing 5.6) takes as input x, the *subject* in question, p, the principal or the *prin*, and proceeds based on whether p is a single subject or a list of subjects. If p is a single subject, s, the *Prop* $x = s$ is returned. Otherwise the disjunction of the translation of the first subject in p (s) and the *rest* of the subjects is returned. The listing 5.6 is the Coq implementation of 4.9.

Listing 5.6: Translation of a 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.
```

A positive translation for a policy (called *trans_policy_positive* in listing 5.7) takes as input e, the *environment*, x, the *subject*, p, the *policy* to translate, $prin_u$, the agreement's user, and a, the asset and proceeds based on whether we have a *PrimitivePolicy* or a *AndPolicy*. If the policy is a *PrimitivePolicy* an implication is returned which indicates x is *permitted* to do *action* to a, if the *preRequisite* holds.

Permitted is a predicate specified as $ParameterPermitted : subject- > act- > asset- > Prop$. So *Permitted* predicate takes a *subject*, an *act* (an action) and an *asset* and builds a term of type *Prop*.

Note that to implement the translation for an *AndPolicy* a local function *trans_p_list* has been defined where for a single *policy*, *trans_policy_positive* is returned, and for a list of *policy*s, the conjunction of *trans_policy_positive*s are returned.

The listing 5.7 is the Coq implementation of 4.13 and 4.14 when $polarity = positive$.

Listing 5.7: Translation of a Positive Policy

```
Fixpoint trans_policy_positive
 (e:environment)(x:subject)(p:policy)(prin_u:prin)(a:asset){struct p} : Prop :=

let trans_p_list := (fix trans_p_list (p_list:nonemptylist policy)(prin_u:prin)(a:
    asset){struct p_list}:=
          match p_list with
           | Single p1 => trans_policy_positive e x p1 prin_u a
           | NewList p p_list' =>
              ((trans_policy_positive e x p prin_u a) /\
               (trans_p_list p_list' prin_u a))
          end) in


 match p with
  | PrimitivePolicy prq policyId action => ((trans_preRequisite e x prq (Single
    policyId) prin_u) ->
                              (Permitted x action a))
```

```
  | AndPolicy p_list ⇒ trans_p_list p_list prin_u a
 end.
```

A negative translation for a policy (called *trans_policy_negative* in listing 5.8) takes as input e, the *environment*, x, the *subject*, p, the *policy* to translate, and a the asset and proceeds based on whether we have a *PrimitivePolicy* or a *AndPolicy*. If the policy is a *PrimitivePolicy* an implication is returned which indicates x is forbidden to do *action* to a regardless of whether *preRequisite* holds. As is the case for the positive translation, to implement the translation for an *AndPolicy* a local function *trans_p_list* has been defined where for a single *policy*, *trans_policy_negative* is returned, and for a list of *policy*s, the conjunction of *trans_policy_negative*s are returned.

The listing 5.8 is the Coq implementation of 4.13 and 4.14 when $polarity = negative$.

Listing 5.8: Translation of a Negative Policy

```
Fixpoint trans_policy_negative
  (e:environment)(x:subject)(p:policy)(a:asset){struct p} : Prop :=
let trans_p_list := (fix trans_p_list (p_list:nonemptylist policy)(a:asset){
    struct p_list}:=
            match p_list with
              | Single p1 ⇒ trans_policy_negative e x p1 a
              | NewList p p_list' ⇒ ((trans_policy_negative e x p a) /\
                                    (trans_p_list p_list' a))
            end) in


  match p with
   | PrimitivePolicy prq policyId action ⇒ not (Permitted x action a)
   | AndPolicy p_list ⇒ trans_p_list p_list a
  end.
```

The translation of a *prerequisite* (called *trans_preRequisite* in listing 5.9) takes as input e, the *environment*, x, the *subject*, *prq*, the *preRequisite* to translate, IDs, the set of identifiers (of policies implied by the *prq*), $prin_u$, the agreement's user, and proceeds by case analysis on the structure of the *prerequisite*. A *prerequisite* is either a $TruePrq$, a $Constraint$, a $ForEachMember$, a $NotCons$, a $AndPrqs$, a $OrPrqs$ or a $XorPrqs$.

In listing 5.9 the translation for $TruePrq$ is the Prop $True$, the translations for $Constraint$, $ForEachMember$ and $NotCons$ simply call respective translation functions for corresponding types *constraint* and $forEachMember$ (namely *trans_constraint*, $trans\_forEachMember$ and *trans_notCons*). Note that the translation for $AndPrqs$, $OrPrqs$ and $XorPrqs$ have not yet been implemented but based on the their many-sorted-logic formulas' specifications (listing 2.12) they will be conjunctions, disjunctions and exclusive disjunctions of translations for each *prerequisite*.

The listing 5.9 is the Coq implementation of 4.15.

Listing 5.9: Translation of a PreRequisite

```
Definition trans_preRequisite
 (e:environment)(x:subject)(prq:preRequisite)(IDs:nonemptylist policyId)(prin_u:
   prin) : Prop :=

 match prq with
   | TruePrq ⇒ True
   | Constraint const ⇒ trans_constraint e x const IDs prin_u
   | ForEachMember prn const_list ⇒ trans_forEachMember e x prn const_list IDs
   | NotCons const ⇒ trans_notCons e x const IDs prin_u
   | AndPrqs prqs ⇒ True
   | OrPrqs prqs ⇒ True
   | XorPrqs prqs ⇒ True
 end.
```

The translation of a *constraint* (called *trans_constraint* in listing 5.10) takes as input e the *environment*, x the *subject*, *const*, the *constraint* to translate, IDs, the set of identifiers (of policies implied by the parent *preRequisite*) and $prin_u$, the agreement's user and proceeds by case analysis on the structure of the *constraint*. A *constraint* is either a *Principal*, a *Count* or a $CountByPrin$. The translation for *Principal* returns the translation function (namely *trans_prin*) for the *prn* (the *prin* that accompanies the *const* constraint). The translation for *Count* and $CountByPrin$ return the translation function *trans_count*. For *Count* the *prin* used is the agreement's user, whereas the *prin* used is the one passed to $CountByPrin$ namely *prn*.

The listing 5.10 is the Coq implementation of 4.17.

Listing 5.10: Translation of a 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 $forEachMember$ (called $trans\_forEachMember$ in listing 5.11) takes as input e the *environment*, x the *subject*, *principals*, the set of subjects that override the agreement's user(s), *const_list* the set of constraints and IDs, the set of identifiers (of policies implied by the parent *preRequisite*).

To implement the translation for a *forEachMember* we start by calling an auxiliary function *process_two_lists* 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 a *forEachMember* translation, the call is "*process_two_lists principals const_list*" which returns a list of pairs of subject and constraint namely *prins_and_constraints*. *prins_and_constraints* is then passed to a locally defined function *trans_forEachMember_-Aux* where for a single pair of subject and constraint *trans_constraint* is called and for a list of pairs of subject and constraints, the conjunction of *trans_constraint*s (for the first pair) and *trans_forEachMember_Aux*s (for the rest of the pairs) are returned.

The listing 5.11 is the Coq implementation of 4.18.

Listing 5.11: Translation of forEachMember

```
Fixpoint trans_forEachMember
      (e:environment)(x:subject)(principals: nonemptylist subject)(const_list:
    nonemptylist constraint)
      (IDs:nonemptylist policyId){struct const_list} : Prop :=

let trans_forEachMember_Aux
 := (fix trans_forEachMember_Aux
      (prins_and_constraints : nonemptylist (Twos subject constraint))
      (IDs:nonemptylist policyId){struct prins_and_constraints} : Prop :=

    match prins_and_constraints with
      | Single pair1 => trans_constraint e x (right pair1) IDs (Single (left pair1))
      | NewList pair1 rest_pairs =>
         (trans_constraint e x (right pair1) IDs (Single (left pair1))) /\
         (trans_forEachMember_Aux rest_pairs IDs)
    end) in

    let prins_and_constraints := process_two_lists principals const_list in
    trans_forEachMember_Aux prins_and_constraints IDs.
```

The translation of a *NotCons* (called *trans_notCons* in listing 5.12) takes as input e the *environment*, x the *subject*, *const*, the *constraint* to translate, IDs, the set of identifiers (of policies implied by the parent *preRequisite*) and $prin_u$, the agreement's user and proceeds to return the negation of *trans_constraint* (see listing 5.10).

The listing 5.12 is the Coq implementation of 4.19.

Listing 5.12: Translation of NotCons

```
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 5.13) takes as input e the *environment*, n the total number of times the subjects mentioned in $prin_u$ (last parameter) may invoke the policies identified by IDs (third parameter).

To implement the translation for a *Count* or a *CountByPrin* we start by calling an auxiliary function *process_two_lists* 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 *trans_ count_ aux*.

trans_count_aux 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 *subject*s, the addition of *get_count* (for the first pair) and *trans_count_aux*s (for the rest of the pairs) is returned.

A local variable $running_total$ has the value returned by *trans_count_aux*. Finally the proposition $running_total < n$ is returned as the translation for a *Count* or a *CountByPrin*.

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).

The listing 5.13 is the Coq implementation of 4.23 and of 4.24.

Listing 5.13: Translation of count

```
Fixpoint trans_count
 (e:environment)(n:nat)(IDs:nonemptylist policyId)
 (prin_u:prin) : Prop :=

 let trans_count_aux
  := (fix trans_count_aux
      (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 rest_pairs)
    end) in

 let ids_and_subjects := process_two_lists IDs prin_u in
 let running_total := trans_count_aux ids_and_subjects in
 running_total < n.
```

Chapter 6

Queries

6.1 Introduction

We first mentioned queries in chapter 4 on page 15. Ultimately policy statements describing an agreement will be used to enforce those agreements. To enforce policy agreements, access queries are asked from the policy engine and access is granted or denied based on the answer.

By defining formal semantics for ODRL the authors of [PW06] were able to prove that answering a query on whether access should be granted or not, is decidable and NP-hard for the full ODRL.

In this chapter we will review our encoding of queries in Coq and Coq representations of other definitions used in [PW06] which we will use to prove decidability results of our own.

6.2 Queries

Queries are tuples of the form $(A, s, action, a, e)$ in [PW06]. The tuple corresponds to the question of determining whether a set A of agreements imply that a subject s may perform action *action* on an asset a given the environment e. The Coq representation is listed in listing 6.1. We distinguish single agreement queries from multiple agreement queries by defining two separate types: *single_query* and *general_query*.

Listing 6.1: Queries

```
Inductive single_query : Set :=
  | SingletonQuery : agreement → subject → act → asset → environment →
    single_query.


Inductive general_query : Set :=
  | GeneralQuery : nonemptylist agreement → subject → act → asset →
    environment → general_query.
```

6.3 Answering Queries

Answering a query as defined earlier can lead to one of four outcomes: error(listing 6.2), permitted(listing 6.3), denied(listing 6.4) and "not applicable" (listing 6.5) defined in [TK06]. Note that e denotes 'an environment being consistent' (see section 5.1) in the following listings.

Listing 6.2: Answerable Queries: Error

$$(\bigwedge[\![agreement]\!]) \wedge e \implies Permitted(s, act, a) \text{ and }$$
$$(\bigwedge[\![agreement]\!]) \wedge e \implies \neg Permitted(s, act, a)$$

Listing 6.3: Answerable Queries: Permit

$$(\bigwedge[\![agreement]\!]) \wedge e \implies Permitted(s, act, a) \text{ and }$$
$$(\bigwedge[\![agreement]\!]) \wedge e \not\Longrightarrow \neg Permitted(s, act, a)$$

Listing 6.4: Answerable Queries: Deny

$$(\bigwedge[\![agreement]\!]) \wedge e \not\Longrightarrow Permitted(s, act, a) \text{ and }$$
$$(\bigwedge[\![agreement]\!]) \wedge e \implies \neg Permitted(s, act, a)$$

Listing 6.5: Answerable Queries: Not Applicable

$$(\bigwedge[\![agreement]\!]) \wedge e \not\Longrightarrow Permitted(s, act, a) \text{ and }$$
$$(\bigwedge[\![agreement]\!]) \wedge e \not\Longrightarrow \neg Permitted(s, act, a)$$

In [PW06] a slightly different formulation is used to denote the same four decision types. "Query Inconsistent","Permission Granted", "Permission Denied" and "Permission Unregulated". They define the formulas f_q^+ and f_q^- as below (see listings 6.6 and 6.7).

Listing 6.6: f_q^+

$$f_q^+ \triangleq (\bigwedge[\![agreement]\!]) \implies Permitted(s, act, a)$$

Listing 6.7: f_q^-

$$f_q^- \triangleq (\bigwedge[\![agreement]\!]) \implies \neg Permitted(s, act, a)$$

Now answering the queries will depend on the *E-validity* of f_q^+ and f_q^-. E-validity or the consistency of the environment is not captured explicitly in [PW06] (see listings 6.8, 6.9, 6.10 and 6.11) however the decision of which answer a query results in takes the consistency of the environment into account. In this thesis we will encode and use the decision algorithms in [PW06].

Listing 6.8: Answerable Queries: Query Inconsistent

f_q^+ `and` f_q^- `both hold`

Listing 6.9: Answerable Queries: Permission Granted

f_q^+ `holds and` f_q^- `does not hold`

Listing 6.10: Answerable Queries: Permission Denied

f_q^+ `does not hold and` f_q^- `holds`

Listing 6.11: Answerable Queries: Permission Unregulated

f_q^+ `does not hold and` f_q^- `does not hold`

Chapter 7

Examples

7.1 Introduction

In this chapter we will take a tour of the syntax and semantics we have so far developed by examining some example agreements. In the following we will start by reviewing some of the examples used in [PW06].

Ultimately the goal of specifying all the syntax and semantics is to declare some interesting theorems about policy expressions and proving them in Coq. However we will first start with some specific propositions/theorems about these examples to get a feel for how proofs are done in Coq.

7.2 Agreement 2.1

Consider example 2.1 (from [PW06]) where the *policySet* is a *AndPolicySet* with $p1$ and $p2$ as the individual *policySet*s. Let $p1$ be defined as $Count[5] \rightarrow print$ and $p2$ as $and[Alice, Count[2]] \rightarrow print$.

The agreement is that the asset *The Report* may be printed a total of five times by either *Alice* or *Bob*, and twice more by Alice. So if Alice and Bob have used policy $p1$ to justify their printing of the asset m_{p1} and n_{p1} times, respectively, then either may do so again if $m_{p1} + n_{p1} < 5$. If they have used $p2$ to justify their printing of the asset m_{p2} and n_{p2} times, respectively, then only Alice may do so again if $m_{p2} + n_{p2} < 2$. Note that since Bob doesn't meet the prerequisite of being Alice, n_{p2} is effectively 0, so we have $m_{p2} < 2$ as the condition for Alice being able to print again (Alice does meet the prerequisite of being Alice).

Listing 7.1: Agreement 2.1 (as used in [PW06])

```
agreement
 for {Alice, Bob}
 about TheReport
 with and [p1, p2].
```

The Coq version of the agreement 2.1 (listing 7.1) and its sub-parts is listed below. It is best to start with the agreement itself called $A2.1$ in the listing and compare to the agreement 2.1 listed in 7.1.

Listing 7.2: Agreement 2.1 in Coq

```
Definition p1A1:policySet :=
 PrimitivePolicySet
  TruePrq
  (PrimitivePolicy (Constraint (Count  5)) id1 Print).

Definition p2A1prq1:preRequisite := (Constraint (Principal (Single Alice))).
Definition p2A1prq2:preRequisite := (Constraint (Count 2)).

Definition p2A1:policySet :=
 PrimitivePolicySet
  TruePrq
  (PrimitivePolicy (AndPrqs (NewList p2A1prq1 (Single p2A1prq2))) id2 Print).

Definition A2.1 := Agreement (NewList Alice (Single Bob)) TheReport
               (AndPolicySet (NewList p1A1 (Single p2A1))).
```

7.3 Agreement 2.5

Consider example 2.5 (from [PW06]) where the *policySet* is a *PrimitivePolicySet* with a *Count* constraint as prerequisite and a *AndPolicy* as the policy. The *AndPolicy* is the conjunction of two *PrimitivePolicy*s. Both policies have prerequisites of type *ForEachMember* with actions *display* and *print* respectively. The *prin* component for both *ForEachMember*s is *Alice*, *Bob*, whereas the constraint for the first *ForEachMember* is *Count*[5] and for the second is *Count*[2].

Listing 7.3: Agreement 2.5 (as used in [PW06])

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

The Coq version of the agreement 2.5 (listing 7.3) and its sub-parts is listed below. See agreement 2.5 listed in 7.3 for comparison.

The agreement is that the asset *ebook* may be displayed up to five times by Alice and Bob each, and printed once by each. However the total number of actions (either *display* or *print*) justified by the two policies by either Alice and Bob is at most 10.

Listing 7.4: Example 2.5

```
Definition tenCount:preRequisite := (Constraint (Count 10)).
Definition fiveCount:constraint := (Count 5).
Definition oneCount:constraint := (Count 1).

Definition prins2_5 := (NewList Alice (Single Bob)).
Definition forEach_display:preRequisite := ForEachMember prins2_5 (Single fiveCount)
    .
Definition forEach_print:preRequisite := ForEachMember prins2_5 (Single oneCount).

Definition primPolicy1:policy := PrimitivePolicy forEach_display id1 Display.
Definition primPolicy2:policy := PrimitivePolicy forEach_print id2 Print.

Definition policySet2_5:policySet :=
 PrimitivePolicySet tenCount (AndPolicy (NewList primPolicy1 (Single primPolicy2))).

Definition A2.5 := Agreement prins2_5 ebook policySet2_5.
```

Chapter 8

Some Simple Theorems

8.1 Introduction

In this chapter we will declare and prove some very simple theorems about the examples from chapter 7. This simple introduction is only meant to give us a feel for how theorems are stated in Coq and how proofs are constructed using Coq *tactics*.

As mentioned earlier, propositions are types in Coq whose type is the sort *Prop*. Any term t whose type is a proposition is a proof term or, for short, a proof. A *Hypothesis* is a local declaration $h : P$ where h is an identifier and P is a proposition. An *Axiom* is similar to a hypothesis except it is declared at the global scope and so it is always available.

A *Theorem* or *Lemma* is stated by giving an identifier whose type is a proposition ([BC04]). The proposition is the statement of the theorem or lemma. It must be followed by a proof. Keywords "Hypothesis", "Axiom" and "Theorem" or "Lemma" are used in each case respectively.

To build a proof in Coq the user states the proposition to prove; this is called a goal to be proved or discharge, along with some hypothesis that makes up the local context. The user then uses commands called tactics to manipulate the local context and to decompose the goal into simpler goals. The goal simplification into sub-goals will continue until all the sub-goals are solved.

In listing 8.1 we have declared a theorem called $example1$ and the corresponding proposition $forallx : nat, x < x + 1$.

Note that the notation $P : T$ is also used to declare program P has type T. This duality of notation is due to Curry-Howard isomorphism which relates the two worlds of type theory and structural logic together [BC04]. Once the Theorem has been declared Coq displays the proposition to be proved under a horizontal line written ———, and displays the context of local facts and hypothesis, if any, above the horizontal line. At this point one can enter proof mode by using *Proof.* upon which Coq is ready to accept tactics. Entering tactics that can break the stated goal (under the horizontal line) into one or more sub-goals is how one progresses until no goals left at which point Coq responds with "No more subgoals" [Ber10].

Listing 8.1: Proof Example

```
Theorem example1: ∀ x:nat, x < x + 1.
```

In the following listings, for the sake of completeness of the presentation, we will include the Coq commands that complete the proof of the respective theorem. We will not however explain the individual commands further here. Showing the commands in the listings, is meant as an indication of the size of the proof in terms of lines of Coq script.

8.2 Theorem One

In listing 8.3 we define a *policySet* with a *constraint* such that if *Alice* has used the policy with *id*1 to justify her printing a_1 times, she may do so again if $a_1 < 5$. The agreement *AgreeCan* simply links the asset *TheReport* with the subject *Alice* and the *policySet* previously defined.

We capture the fact that *Alice* has used the policy with *id*1 to justify her printing 2 times in an environment called *eA*1. Recall that environments are defined to be non-empty lists of $count_equality$ objects (see listing 5.1).

We also declare a hypothesis H with the proposition that results from the translation of the agreement (see definition of $trans_agreement$ in listing 5.4) and the environment. The proposition can be shown in Coq after some clean-up (e.g. replaced 101 by Alice) and using the form *Eval compute* :

Listing 8.2: Hypothesis for Theorem One

```
∀ x : subject, x = Alice /\ True → 2 < 5 → Permitted x Print TheReport.
```

The theorem *One* that we are going to prove is trivial but nonetheless in English it states that Alice is Permitted to Print TheReport. The proof comes after the command 'Proof.' and ends with 'Qed'.

Listing 8.3: Theorem One

```
Definition psA1:policySet :=
 PrimitivePolicySet
  TruePrq
  (PrimitivePolicy (Constraint (Count  5)) id1 Print).

Definition AgreeCan := Agreement (Single Alice) TheReport psA1.

Definition eA1 : environment :=
 (SingleEnv (make_count_equality Alice id1 2)).

Hypothesis H: trans_agreement eA1 AgreeCan.

Theorem One: Permitted Alice Print TheReport.
```

```
Proof. simpl in H. apply H. split. reflexivity. auto.  omega. Qed.
```

8.3 Theorem Two

In listing 8.5 we define an exclusive policy set *policySet* containing a policy *pol* that allows printing. The agreement *AgreeA5* includes the exclusive policy set to express that Bob may print *LoveAndPeace*. However any subject that is not the agreement's user (e.g. Bob) is forbidden from printing *LoveAndPeace*.

Notice that due to the fact that environments are defined as non-empty lists, we have added a Null count to it (see $eA5$). We continue to capture the relevant facts from the environment and the agreement through defining a hypothesis (e.g. H). The hypothesis is shown in listing 8.5.

Listing 8.4: Hypothesis for Theorem Two

```
∀ x : subject,
      (x = Bob /\ True → True → Permitted x Print LoveAndPeace) /\
     ((x = Bob → False) → Permitted x Print LoveAndPeace → False).
```

Theorem $T1\_A5$ states the exclusivity of the policy set, namely that any subject that is not Bob is not permitted to print the asset LoveAndPeace. Theorem $T2\_A5$ uses $T1\_A5$ to prove Alice is not permitted to print the asset.

Listing 8.5: Theorem Two

```

Definition prin_bob := (Single Bob).
Definition pol:policy := PrimitivePolicy TruePrq id3 Print.
Definition pol_set:policySet := PrimitiveExclusivePolicySet TruePrq pol.
Definition AgreeA5 := Agreement prin_bob LoveAndPeace pol_set.
Definition eA5 : environment := (SingleEnv (make_count_equality NullSubject NullId 0))
    .



Hypothesis H: trans_agreement eA5 AgreeA5.



Theorem T1_A5: ∀ x, x<>Bob →  ~Permitted x Print LoveAndPeace.
Proof. simpl in H. apply H. Qed.

Theorem T2_A5: ~Permitted Alice Print LoveAndPeace.
Proof. simpl in H. apply T1_A5. apply not_eq_S. omega. Qed.



End A5.
```

8.4 Theorem Three

In listing 5.1 we defined environments as non-empty lists of $count_equality$ objects which are in turn defined as counts per each subject, policy-id pair. These count formulas represent how many times each policy has been used to justify an action by a subject wrt a policy (specified by the policy id) and semantically it makes sense that they are unique in time. When and if two $count_equality$ objects with the same subject and policy id refer to different counts, we say we have *inconsistent* count formulas. The listing 8.6 defines the binary predicate *inconsistent*.

Listing 8.6: Inconsistent Count Formulas

```
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.
```

Next we would like to expand the notion of inconsistency to more than two count formulas. We first define a predicate over a count formula and an environment as in listing 8.7. If the environment is a singleton then we just compare the two count formulas for inconsistency, else we build the disjunction of the inconsistency between the count formula on one hand and the head of the environment and the rest of the environment, respectively.

Listing 8.7: Inconsistent Count Formula And Environment

```
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.
```

Finally we define a new inductive data type that represents *consistent* environments (see listing 8.8). An environment is consistent if it is a singleton count formula, if it consists of only two consistent count formulas and finally if the environment consists of a consistent environment and the consistent composition of a count formula and the consistent environment (see constructor *consis_more*.

Listing 8.8: Inconsistent Environment

```
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).
```

We will now pose several small theorems about consistency of count formulas and environments and provide proofs for them (see listing 8.9).

Listing 8.9: Inconsistent Environment

```
Theorem f1_and_f2_are_inconsistent: inconsistent f1 f2.
Proof.
unfold inconsistent. simpl. omega. Qed.


Theorem f1_and___env_of_f2_inconsistent: formula_inconsistent_with_env f1 (
   SingleEnv f2).
Proof.
unfold formula_inconsistent_with_env. apply f1_and_f2_are_inconsistent. Qed.


Theorem two_inconsistent_formulas_imply_env_inconsistent:
 ∀ f g, inconsistent f g → ~env_consistent (ConsEnv f (SingleEnv g)).
Proof.
intros. unfold not. intros H'.
inversion H'. intuition. intuition. Qed.


Theorem e2_is_inconsistent: ~env_consistent e2.
Proof.
apply two_inconsistent_formulas_imply_env_inconsistent.
apply f1_and_f2_are_inconsistent. Qed.


Theorem env_consistent_implies_two_consistent_formulas:
 ∀ (f g: count_equality),
   env_consistent (ConsEnv f (SingleEnv g))→ ~inconsistent f g.
Proof.
intros. inversion H. exact H1. intuition. Qed.


Theorem two_consistent_formulas_imply_env_consistent:
 ∀ (f g: count_equality),
   ~inconsistent f g → env_consistent (ConsEnv f (SingleEnv g)).
Proof.
intros. apply consis_2. exact H. Qed.

Theorem env_inconsistent_implies_two_inconsistent_formulas:
 ∀ (f g: count_equality),
   ~env_consistent (ConsEnv f (SingleEnv g))→ inconsistent f g.
```

```
Proof.
induction f.
induction g.
unfold inconsistent.
intros.
subst.
generalize (dec_eq_nat n n0).
intro h; elim h.
intro; subst.
elim H.
apply consis_2.
unfold inconsistent.
intro.
assert (s0=s0); auto.
assert (p0=p0); auto.
specialize H0 with (1:=H1) (2:=H2).
elim H0; auto.
auto.
Qed.


Theorem same_subjects_policyids_different_counts_means_inconsistent : ∀ (s1 s2:
    subject),
           ∀ (id1 id2: policyId),
           ∀ (n1 n2: nat),

  (s1 = s2 /\ id1 = id2 /\ n1 <> n2) →
  inconsistent (CountEquality s1 id1 n1) (CountEquality s2 id2 n2).

Proof.
intros. unfold inconsistent. intros. intuition. Qed.
```

Chapter 9

Proposed Future Work

9.1 Summary

We started off by looking at DRELs and specifically at ODRL and considered its formal semantics as captured by [PW06]. We presented the encodings and semantics of the constructs for a significant subset of ODRL in Coq and then defined what queries looked like and what the decision problem was in this context. We have also encoded the decision algorithms as presented in [PW06] in Coq in order to perform formal verification of theorems of interest. We noted the common thread between RELs and policy languages for access control systems such as those between ODRL and SELinux. We discussed the goal of generalizing the concept of a policy language from strictly representing subsets of ODRL, to representing (subsets of) both ODRL and SELinux policy languages with the goal of applying the decision algorithms to both types of policies, in a unified manner.

9.2 Machine-Checked Proof of Decidability of Queries

By defining formal semantics for ODRL authors of [PW06] were able to show some important results. First result is that answering the question of whether a set of ODRL statements imply a permission, denial or other possibilities is decidable and also that its complexity is NP-hard.

The authors of [PW06] then prove that by removing the construct $not[policySet]$ from ODRL's syntax answering the same query remains decidable and efficient (polynomial time complexity).

We will prove equivalent results as above starting with the decidability result of answering a query in ODRL0 (which does not include $not[policySet]$). The theorem in listing 9.1 states that for all environments, all single agreements, all subjects, all actions and all assets, either permission is granted, permission is denied, permission is unregulated or query is inconsistent.

Listing 9.1: Environments and Counts

```
Theorem queriesAreDecidable: ∀ (e:environment),
          ∀ (agr: agreement),
          ∀ (s:subject),
          ∀ (action:act),
          ∀ (a:asset),

(permissionGranted e [agr] s action a) \/
(permissionDenied e [agr] s action a)  \/
(queryInconsistent e [agr] s action a) \/
(permissionUnregulated e [agr] s action a).
```

We will then augment ODRL0 with the constructs we omitted from the full ODRL (resulting in what we have earlier called ODRL1 or ODRL2) including the troublesome construct *not*[*policySet*] and attempt to prove that the decidability results remain intact. There is a chance that a proof is not possible due to particulars of the Coq encoding we have used, in in which case, we will adjust our encoding.

9.3 SELinux

We started out by looking at DRELs and specifically ODRL where rights expressions are used to arbitrate access to assets under conditions. Recall that the main construct in ODRL is the agreement which specifies users, asset(s) and policies (as part of policy sets) whereby controls on users' access to the assets are described. This is reminiscent of how access control conditions are expressed in access control policy languages such as XACML and SELinux.

While XACML is a high-level and platform independent access control system SELinux 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 role-based access control (RBAC). 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 user space by a system call, 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].

9.4 SELinux Policy Language

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 a type, a role and a type. First types are declared using a type declaration statement. 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 9.2). Note that in the listings below we will present a simplified and modified version of the official SELinux syntax in order to (re-)use the ODRL Coq framework.

Listing 9.2: Declarations

```
'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 9.3) specify which operations are allowed and whether to audit (log). Any operation not covered by AV rules are denied 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* (with avkind=*deny* meaning not to allow).

Listing 9.3: 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* 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 9.4).

Listing 9.4: 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 9.5. Whenever a permission is requested on an object class C, the security server checks that the constraints hold.

Listing 9.5: Constrain Rule

```
<constraint> → 'constrain' C, P, <expr> ';'

<expr> → 'not' <expr> | <expr> 'and' <expr> | <expr> or <expr> | U1 <
   op> U2 | T1 <op> T2 | R1 <op> R2

<op> → '==' | '!='
```

9.5 Agreements in SELinux

As with ODRL we will start by limiting the policy language to only allow AV rules. As mentioned earlier an operation not covered by a allow rule is denied by SELinux. We will make up explicit *deny* rules therefore an agreement is defined to be a combination of allow and deny rules. Allow and deny rules as mappings are defined in listing 9.6.

Listing 9.6: 'allow'/'deny' Rule as a Mapping

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

Listing 9.7: SELinux Agreement

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

9.6 Environments

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.

9.7 Queries in SELinux

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 equivalent to the *Permitted* predicate we saw earlier for ODRL.

Listing 9.8: f_q^+ for SELinux

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

Listing 9.9: f_q^- for SELinux

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

9.8 Decidability of Queries in SELinux

In this thesis we will be investigating the question of decidability for answering queries in SELinux policies based on the same four outcomes we encountered earlier in 6.3 namely error, permitted, denied and "not applicable". We will first state a decidability theorem similar to the theorem in listing 9.1 (minor adjustments may be needed to allow for differences with SELinux policy language) and present a proof for it in Coq. The literature in the SELinux implies only two outcomes are possible: permitted or denied. We will next attempt to prove this conjecture in Coq. Finally we will add constraint relations (see listing 9.10 and listing 9.11) to SELinux policies (which we have not included so far) and prove the same decidability results again for the augmented policy.

Listing 9.10: f_q^+ for SELinux

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

Listing 9.11: f_q^- for SELinux

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

References

[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.

[Ber10] Yves Bertot. Coq in a hurry. Technical report, May 2010. http://cel.archives-ouvertes.fr/inria-00001173/PDF/coq-hurry.pdf.

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

[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.

[HW08] 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.

[MRB98] Robert Martin, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design 3*, volume 256. Addison-Wesley, 1998.

[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.

[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.

[SM12] Michel St-Martin. *A verified algorithm for detecting conflicts in XACML access control rules*. PhD thesis, Carleton University, 2012.

[SMF13] Michel St-Martin and Amy Felty. A verified algorithm for detecting conflicts in XACML access control rules. In *Informal Proceedings of the First International Workshop on Automated Reasoning in Security and Software*, pages 4–11, 2013. http://www.site.uottawa.ca/~afelty/dist/arsec13.pdf.

[SS09] Nicholas Paul Sheppard and Reihaneh Safavi-Naini. On the operational semantics of rights expression languages. In 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*, pages 17–28. ACM, 2009.

[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].

[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.