

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

This is the abstract.

Acknowledgements

I would like to thank all the little people who made this possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables	viii
List of Figures	ix
1 Introduction	1
2 Middle	3
2.1 Logic Based Semantics for ODRL	3
2.2 Pucella 2006	3
2.3 what will I do?	4
2.3.1 Coq	4
3 ODRL0 Syntax	5
3.1 Introduction	5
3.2 ODRL0	6
3.3 Productions	7
4 ODRL0 Syntax In Coq	10
5 ODRL0 Semantics	13
5.1 Introduction	13
5.2 Agreement Translation	13
5.3 Policy Set Translation	14
5.3.1 PrimitivePolicySet Translation	14
5.3.2 PrimitiveExclusivePolicySet Translation	14
5.3.3 AndPolicySet Translation	15
5.4 Principal Translation	15

5.4.1	Single Subject Translation	15
5.4.2	List of Subjects Translation	16
5.5	Prerequisite Translation	16
5.5.1	True Prerequisite Translation	16
5.5.2	Constraint Prerequisite Translation	16
5.5.3	ForEachMember Prerequisite Translation	16
5.5.4	NotCons Prerequisite Translation	17
5.5.5	AndPrqs Prerequisite Translation	17
5.5.6	OrPrqs Prerequisite Translation	17
5.5.7	XorPrqs Prerequisite Translation	17
5.6	Constraint Translation	18
5.6.1	Principal Constraint Translation	18
5.6.2	Count Constraint Translation	18
5.6.3	CountByPrin Constraint Translation	18
5.6.4	forEachMember Translation	19
5.6.5	"Not Constraint" Translation	19
5.7	Count Translation	19
5.7.1	Count Translation For Subject/ID Pair	19
5.7.2	Count Translation For Subject/ID Pairs	20
6	ODRL0 Semantics In Coq	21
6.1	Introduction	21
6.2	Translations	21
7	Queries	28
7.1	Introduction	28
7.2	Translations	28
8	Examples	35
8.1	Introduction	35
8.2	Agreement 2.1	35
8.3	Agreement 2.5	36
8.4	Agreement 2.6	37

9	Some Simple Theorems	38
9.1	Introduction	38
9.2	Theorem One	39
9.3	Theorem Two	40
9.4	Theorem Three	41
10	Proposed Future Work	44
10.1	Introduction	44
10.2	SELinux	45
10.3	SELinux Policy Language	45
10.4	Authorization Relation	46
	APPENDICES	48
	References	48

List of Tables

List of Figures

Chapter 1

Introduction

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 whitepaper “A digital rights management ecosystem model for the education community,” digital rights management systems cover the following four areas: 1) defining rights 2) distributing/acquiring rights 3) enforcing rights and 4) tracking usage [4].

Rights Expression Languages (REL)s, or more precisely when dealing with digital assets, Digital Rights Expression Languages (DREL)s 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) [bib], and the Open Digital Rights Language (ODRL)[bib]. 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*, are expressed [wikipedia]. 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 [problem with RELs paper] in *Fair Play* (Apple) and in *PlayReady* (Microsoft). The authors of [the problem with RELs] argue that both these RELs and other ones are simply too complex to be used effectively since they try to cover much of the DRM ecosystem.

Another issue with the current batch of RELs are due to their semantics being expressed in a natural language (e.g. English). By necessity natural languages are ambiguous and open to interpretation.

To formalize the semantics of RELs several approaches have been attempted by various authors. The main categories are logic based, operational semantics based interpreters and finally web ontology based (from the Knowledge Representation Field). In this thesis we will focus on the logic based approach to formalizing semantics and will study a specific logic based language that is a translation from a subset of ODRL.

Chapter 2

Middle

2.1 Logic Based Semantics for ODRL

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. However propositional logic 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 called “Predicate Logic” or “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 [Using First Order Logic to Reason about Policies] 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.

2.2 Pucella 2006

Pucella and Weissman [6] specify a predicate logic based language that represents a subset of ODRL.

2.3 what will I do?

2.3.1 Coq

• Program correctness • Formal verification of software • Certified programs • Proof assistant • Interactive and mechanized theorem proving • Examples of machine assisted proofs: CompCert, four-color theorem proof • Coq is based on a higher-order functional programming language • Dependent Types • Subset types • Easier than writing explicit proofs • Write formal specification and proofs that programs comply to their specification (a-short-intro-to-coq) • Automatically extract code from specifications as Ocaml or Haskell (a-short-intro) • Properties, programs and proofs are all formalized in the same language called CIC (Calculus of inductive Constructions). (a-short-intro) • Coq uses a sort called Prop for propositions • Coq art: • Well-formed propositions are assertions one can express about values such as mathematical objects or even programs e.g. $3 < 8$ • Note that assertions may be true, false or simply conjectures • An assertion is only true in general if a proof is provided • However hand written proofs are difficult to verify • Coq provides an environment for developing proofs including a formal language to express proofs in, the language itself being built using proof theory making it possible to step by step verification of the proofs • Mechanized proof verification requires a "proof" that the verification algorithm is correct itself in applying all the formal rules correctly

Chapter 3

ODRL0 Syntax

3.1 Introduction

Authors of [6] 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 and the equivalent one expressed in the abstract syntax [6].

Listing 3.1: agreement for Mary Smith in 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 3.1 is shown below using the syntax from [6].

Listing 3.2: agreement for Mary Smith as BNF (as used in [6])

```

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).

3.2 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. For example, *Alice* may satisfy the requirement “The user must pay 5 dollars”. Finally conditions are constraints that must not hold.

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 3.12). Note that we have also removed the condition *not[policySet]* from

ODRL since the authors in [6] 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 (except for *not[policySet]*) 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. BNF style grammars are less formal as they give some suggestions about the surface syntax of expressions [Pierce1] without getting into lexical analysis and parsing related aspects such as precedence order of operators. 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 [Pierce1]. To get started let’s see what the listing 3.2 would look like in ODRL0’s Coq version.

Listing 3.3: 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))))).
```

3.3 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 3.4: 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 for the application we will be using throughout.

Listing 3.5: prin

```
<prin> ::= { <subject1>, ..., <subjectm> }
```

Listing 3.6: subject

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


Assets are also application specific but similar to subjects we will use specific ones for the DRM application (taken from [6]). *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 3.7: 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 3.8: policySet

```
<policySet> ::=
  <preRequisite> → <policy>                ; primitive policy set
  <preRequisite> ⇨ <policy>                ; primitive exclusive policy set
  'and' [ <policySet1>, ..., <policySetm> ] ; 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 3.9: policy

```
<policy> ::=
  <preRequisite> ⇒<policyId> <act>          ; primitive policy
  'and' [ <policy1>, ..., <policym> ]      ; conjunction
```

An *Action* (*act*) is represented as a natural number. Similar to assets and subjects, actions are application specific. Some example actions taken from [6] are *Display* and *Print*.

Listing 3.10: act

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

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

Listing 3.11: 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 ?? section. 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 3.12: preRequisite

```
<preRequisite> ::=
  'True'                                ; always true
  <constraint>                          ; constraint
  'ForEachMember' [<prin> ; <constraint1>, ..., <constraintm> ]
  ; constraint distribution
  'not' [ <constraint> ]                ; suspending constraint
  'and' [ <preRequisite1>, ..., <preRequisitem> ]
  ; conjunction
  'or' [ <preRequisite1>, ..., <preRequisitem> ]
  ; disjunction
  'xor' [ <preRequisite1>, ..., <preRequisitem> ]
  ; 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 3.13: constraint

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

Chapter 4

ODRL0 Syntax In Coq

ODRL0 productions were presented as high level abstract syntax in 3.3. 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 *subjects* (see 4.1).

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” [5]), 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 3.8. There are three ways a *policySet* can be constructed (see 3.8) 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 *policySets*.

A *policy* is defined as a datatype with constructors *PrimitivePolicy* and *AndPolicy* (see 3.9). *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 *policyIds* 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 4.2) of *policys*.

The data type *nonemptylist* reflects the definition of “policy conjunction” in listing 3.9 in chapter 3 (see definition of *nonemptylist* in listing 4.2). Essentially *nonemptylist*

represents a list data structure that has at least one element and it is defined as a new *polymorphic* inductive type in its own section.

Listing 4.1: 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 4.2: nonemptylist type

```

Section nonemptylist.

Variable X : Set.

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

End nonemptylist.

```

In listing 4.3 *preRequisite* is defined as a new datatype with constructors *TruePrq*, *Constraint*, *ForEachMember*, *NotCons*, *AndPrqs*, *OrPrqs* and *XorPrqs* (see 3.12 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 7.10). 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 3.13 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 4.3: 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.

```

Chapter 5

ODRL0 Semantics

5.1 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. Note that in the listings in this chapter we use $\llbracket \cdot \rrbracket$ (double square brackets) notation as a mapping of ODRL0 syntactic elements to their translations as many-sorted first-order logic formulas. We also use \triangleq between a translation and its corresponding formula to mean the translation is “mapped to” the formula.

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.

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 .

5.2 Agreement Translation

The translation of an *agreement* returns the translation for a *policySet* with arguments $prin_u$, the agreement’s user and a , the asset.

Listing 5.1: Agreement Translation

$\llbracket agreement \text{ for } prin_u \text{ about } a \text{ with } policySet \rrbracket \triangleq \llbracket policySet \rrbracket^{prin_u, a}$

5.3 Policy Set Translation

The translation for a *policySet* ($\llbracket policySet \rrbracket^{prin_u, a}$) is defined by cases, one for each clause of the grammar in listing 3.8. Recall that a *policySet* is either a *PrimitivePolicySet*, a *PrimitiveExclusivePolicySet* or a *AndPolicySet*. Each of these has its own translation function, which will be defined in the next 3 subsections.

5.3.1 PrimitivePolicySet Translation

Translation of a *PrimitivePolicySet* ($preRequisite \rightarrow policy$) yields a formula that includes a test on whether the subject is in the set of agreements' users, 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 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.

Listing 5.2: Policy Set Translation : PrimitivePolicySet

$$\llbracket preRequisite \rightarrow policy \rrbracket^{e, prin_u, a} \triangleq \forall x ((\llbracket prin_u \rrbracket_x \wedge \llbracket preRequisite \rrbracket_x^{e, getId(p), prin_u, a}) \rightarrow \llbracket policy \rrbracket_x^{positive, e, prin_u, a})$$

Listing 5.3: Positive Policy Translation : Single policy

$$\llbracket preRequisite \Rightarrow_{policyId} act \rrbracket_x^{positive, e, prin_u, a} \triangleq (\llbracket preRequisite \rrbracket_x^{e, policyId, prin_u}) \Rightarrow Permitted(x, \llbracket act \rrbracket, a)$$

If the policy is a *AndPolicy*, the translation yields a conjunction of positive translations of each policy in turn.

Listing 5.4: Positive Policy Translation : List of policies

$$\llbracket and[policy_1, \dots, policy_m] \rrbracket^{positive, e, prin_u, a} \triangleq \llbracket policy_1 \rrbracket^{positive, e, prin_u, a} \wedge \dots \wedge \llbracket policy_m \rrbracket^{positive, e, prin_u, a}$$

5.3.2 PrimitiveExclusivePolicySet Translation

PrimitiveExclusivePolicySet ($preRequisite \mapsto policy$) 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. 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.

Listing 5.5: Policy Set Translation : PrimitiveExclusivePolicySet

$$\begin{aligned} \llbracket preRequisite \mapsto policy \rrbracket^{e, prin_u, a} &\triangleq \forall x ((\llbracket prin_u \rrbracket_x \wedge \\ &\llbracket preRequisite \rrbracket_x^{e, getId(p), prin_u, a}) \rightarrow \llbracket policy \rrbracket_x^{positive, e, prin_u, a}) \wedge \forall x \\ &(\neg \llbracket prin_u \rrbracket_x \rightarrow \llbracket policy \rrbracket_x^{negative, e, a}) \end{aligned}$$

Listing 5.6: Negative Policy Translation : Single policy

$$\begin{aligned} \llbracket preRequisite \Rightarrow_{policyId} act \rrbracket_x^{negative, e, prin_u, a} &\triangleq \\ (\llbracket preRequisite \rrbracket_x^{e, policyId, prin_u}) &\Rightarrow \neg(Permitted(x, \llbracket act \rrbracket, a)) \end{aligned}$$

If the policy is a *AndPolicy*, the translation yields a conjunction of negative translations of each policy in turn.

Listing 5.7: Negative Policy Translation : List of policies

$$\llbracket and[policy_1, \dots, policy_m] \rrbracket^{negative, e, a} \triangleq \llbracket policy_1 \rrbracket^{negative, e, a} \wedge \dots \wedge \llbracket policy_m \rrbracket^{negative, e, a}$$

5.3.3 AndPolicySet Translation

AndPolicySet translates to conjunctions of the corresponding policy set translations.

Listing 5.8: Policy Set Translation : AndPolicySet

$$\begin{aligned} \llbracket and[policySet_1, \dots, policySet_m] \rrbracket^{e, prin_u, a} &\triangleq \llbracket policySet_1 \rrbracket^{e, prin_u, a} \wedge \dots \wedge \\ \llbracket policySet_m \rrbracket^{e, prin_u, a} & \end{aligned}$$

5.4 Principal Translation

Translation for a *prin* ($\llbracket prin \rrbracket_x$) 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 ($\{subject_1, \dots, subject_m\}$) so the translation covers both cases. Each of these has its own translation function, which will be defined in the following 2 subsections.

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*.

5.4.1 Single Subject Translation

Listing 5.9: Prin Translation : Single subject

$$\llbracket subject \rrbracket_x \triangleq x = subject$$

5.4.2 List of Subjects Translation

Translation of a list of subjects is the disjunction of the translations for each subject.

Listing 5.10: Prin Translation : List of subjects

$$\llbracket \{subject_1, \dots, subject_m\} \rrbracket_x \triangleq \llbracket subject_1 \rrbracket_x \vee \dots \vee \llbracket subject_m \rrbracket_x$$

5.5 Prerequisite Translation

Translation for a prerequisite is a formula $\llbracket prerequisite \rrbracket_x^{[id_1, \dots, id_m], prin, a}$, where the set of *ids* refer to identifiers for policies that are implied by the prerequisites, *prin* is the agreement's user(s) (and to which the prerequisites apply), *a* is the asset and *x* is a variable of type *subject*. The translation for a *prerequisite* is described by translation formulas for each type of *prerequisite*. A *prerequisite* is either always *true*, a *Constraint*, a *ForEachMember*, a *NotCons*, a *AndPrqs*, a *OrPrqs* or a *XorPrqs*. Each of these has its own translation function, which will be defined in the following subsections.

5.5.1 True Prerequisite Translation

The translation for a *TruePrq* yields a formula that is always *true*.

Listing 5.11: Prerequisite Translation : Always True Prerequisite

$$\llbracket prerequisite :: true \rrbracket \triangleq \text{True}$$

5.5.2 Constraint Prerequisite Translation

The translation for a *Constraint* is handled by a specialized constraint translation function (coverage of which starts at 5.18).

Listing 5.12: Prerequisite Translation : Constraint

$$\llbracket prerequisite :: constraint \rrbracket_x^{[id_1, \dots, id_m], prin_u} \triangleq \llbracket constraint \rrbracket_x^{[id_1, \dots, id_m], prin_u}$$

5.5.3 ForEachMember Prerequisite Translation

The translation for a *ForEachMember* is also handled by a specialized translation function (covered at 5.21).

Listing 5.13: Prerequisite Translation : ForEachMember

$$\llbracket prerequisite :: forEachMember \rrbracket_x^{[subject_1, \dots, subject_k], [constraint_1, \dots, constraint_m], [id_1, \dots, id_n]} \triangleq \llbracket forEachMember \rrbracket_x^{[subject_1, \dots, subject_k], [constraint_1, \dots, constraint_m], [id_1, \dots, id_n]}$$

5.5.4 NotCons Prerequisite Translation

The translation for a *NotCons* yields a formula that is simply the negation of the translation for a constraint.

Listing 5.14: Prerequisite Translation : Not Constraint

$$\llbracket not prerequisite :: constraint \rrbracket_x^{[id_1, \dots, id_m], prin_u} \triangleq \neg \llbracket constraint \rrbracket_x^{[id_1, \dots, id_m], prin_u}$$

5.5.5 AndPrqs Prerequisite Translation

The translation for a *AndPrqs* yields a formula that is the conjunction of the translation for each *preRequisite*.

Listing 5.15: Prerequisite Translation : Conjunction

$$\llbracket and [preRequisite_1, \dots, preRequisite_k] \rrbracket^{[id_1, \dots, id_m], prin_u} \triangleq \llbracket preRequisite_1 \rrbracket^{[id_1, \dots, id_m], prin_u} \wedge \dots \wedge \llbracket preRequisite_k \rrbracket^{[id_1, \dots, id_m], prin_u}$$

5.5.6 OrPrqs Prerequisite Translation

The translation for a *OrPrqs* yields a formula that is the inclusive disjunction of the translation for each *preRequisite*.

Listing 5.16: Prerequisite Translation : Inclusive Disjunction

$$\llbracket or [preRequisite_1, \dots, preRequisite_k] \rrbracket^{[id_1, \dots, id_m], prin_u} \triangleq \llbracket preRequisite_1 \rrbracket^{[id_1, \dots, id_m], prin_u} \vee \dots \vee \llbracket preRequisite_k \rrbracket^{[id_1, \dots, id_m], prin_u}$$

5.5.7 XorPrqs Prerequisite Translation

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

Listing 5.17: Prerequisite Translation : Exclusive Disjunction

$$\llbracket Xor \ [preRequisite_1, \dots, preRequisite_k] \rrbracket_{x}^{[id_1, \dots, id_m], prin_u} \triangleq \llbracket preRequisite_1 \rrbracket_{x}^{[id_1, \dots, id_m], prin_u} \oplus \dots \oplus \llbracket preRequisite_k \rrbracket_{x}^{[id_1, \dots, id_m], prin_u}$$

5.6 Constraint Translation

Translation for a constraint is a formula $\llbracket constraint \rrbracket_x^{[id_1, \dots, id_m], prin_u, a}$, where the set of *ids* refer to identifiers for policies that are implied by the constraint, *prin_u* is the agreement's user(s) (and to which the constraint applies), *a* is the asset and *x* is a variable of type *subject*. The translation for a *constraint* is described by translation formulas for each type of *constraint*. A *constraint* is either a *Principal*, a *Count*, or a *CountByPrin*. Each of these has its own translation function, which will be defined in the following subsections.

5.6.1 Principal Constraint Translation

The translation for a *Principal* is handled by a specialized translation function (covered at 7.4).

Listing 5.18: Constraint Translation : Principal

$$\llbracket constraint :: prin \rrbracket_x^{[subject_1, \dots, subject_m]} \triangleq \llbracket prin \rrbracket_x^{[subject_1, \dots, subject_m]}$$

5.6.2 Count Constraint Translation

The translation for a *Count* is handled by a specialized translation function (covered at ??).

Listing 5.19: Constraint Translation : Count

$$\llbracket constraint :: count[N] \rrbracket_x^{[id_1, \dots, id_m], prin_u} \triangleq \llbracket count[N] \rrbracket_x^{[id_1, \dots, id_m], prin_u}$$

5.6.3 CountByPrin Constraint Translation

The translation for a *CountByPrin* is handled by the same specialized translation function as that for *Count*. The difference is that *CountByPrin* overrides the subjects in *prin_u* by a different set of subjects (covered at ??).

Listing 5.20: Constraint Translation : Count by Principal

$$\llbracket \text{constraint} :: \text{prin}(\text{count}[N]) \rrbracket_x^{[subject_1, \dots, subject_m], [id_1, \dots, id_n]} \triangleq \llbracket \text{prin}(\text{count}[N]) \rrbracket_x^{[subject_1, \dots, subject_m], [id_1, \dots, id_n]}$$

5.6.4 forEachMember Translation

Listing 5.21: ForEachMember Translation : Count by Principal

$$\begin{aligned} \llbracket \text{forEachMember} \rrbracket_x^{[subject_1, \dots, subject_k], [constraint_1, \dots, constraint_m], [id_1, \dots, id_n]} &\triangleq \\ &\llbracket \text{constraint} \rrbracket_x^{(subject_1, constraint_1), [id_1, \dots, id_n]} \wedge \dots \wedge \llbracket \text{constraint} \rrbracket_x^{(subject_1, constraint_m), [id_1, \dots, id_n]} \\ &\wedge \dots \wedge \llbracket \text{constraint} \rrbracket_x^{(subject_2, constraint_1), [id_1, \dots, id_n]} \wedge \dots \wedge \\ &\llbracket \text{constraint} \rrbracket_x^{(subject_2, constraint_m), [id_1, \dots, id_n]} \wedge \dots \wedge \llbracket \text{constraint} \rrbracket_x^{(subject_k, constraint_1), [id_1, \dots, id_n]} \\ &\wedge \dots \wedge \llbracket \text{constraint} \rrbracket_x^{(subject_k, constraint_m), [id_1, \dots, id_n]} \end{aligned}$$

5.6.5 "Not Constraint" Translation

The translation for "Not Constraint" was listed in listing 5.14 earlier but we repeat it here to go along the Coq version.

Listing 5.22: Not Constraint Translation

$$\llbracket \text{not constraint} \rrbracket_x^{[id_1, \dots, id_m], \text{prin}_u} \triangleq \neg \llbracket \text{constraint} \rrbracket_x^{[id_1, \dots, id_m], \text{prin}_u}$$

5.7 Count Translation

Translation for *Count* or *CountByPrin* is based on whether the translation is on a single pair and multiple pairs of subject/policyIds. Each of these two cases will be described in the following subsections.

5.7.1 Count Translation For Subject/ID Pair

The translation for *Count* or *CountByPrin* for a pair of subject and policy identifier is a formula that is true if the number of times the *subject*₁ has invoked a policy with policy identifier *id*₁ is smaller than *N*.

Listing 5.23: Count Translation : subject and policyId pair

$$\llbracket \text{count}[N] \rrbracket_x^{subject_1, id_1} \triangleq \text{getCount}(subject_1, id_1) < N$$

5.7.2 Count Translation For Subject/ID Pairs

The translation for *Count* or *CountByPrin* for subject and policy identifier pairs is a formula that is true if the total number of times that a subject has invoked a policy with policy identifier id_i is smaller than N .

Listing 5.24: Count Translation : subject and policyId pairs

$$\begin{aligned} \llbracket count[N] \rrbracket_x^{[id_1, \dots, id_m], prn} &\triangleq \\ (getCount(getSubject(prn)_1, id_1) + \dots + &getCount(getSubject(prn)_1, id_m) + \dots + \\ &getCount(getSubject(prn)_k, id_1) + \dots + &getCount(getSubject(prn)_k, id_m)) < N \end{aligned}$$

Chapter 6

ODRL0 Semantics In Coq

6.1 Introduction

The translation functions plus the auxiliary types and infrastructure, implementing the semantics have been encoded in Coq. Translation functions all build terms of *sort Prop*.

Talk about Props here...

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 ?? 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.

Listing 6.1: Environments and Counts

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

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

6.2 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*.

Listing 6.2: Translation of agreement

```

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

```

Translation of a *policySet* (called *trans_ps* in listing 7.3), 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_{pslist}* has been defined where for a single *policySet*, *trans_{ps}* is called, and for a list of *policySets*, the conjunction of *trans_{ps}* are returned.

Listing 6.3: 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  $\Rightarrow$  trans_ps e ps1 prin_u a
  | NewList ps ps_list'  $\Rightarrow$  ((trans_ps e ps prin_u a) /\ (trans_ps_list ps_list' prin_u a
  ))
  end) in
  match ps with
  | PrimitivePolicySet prq p  $\Rightarrow$   $\forall$  x, (((trans_prin x prin_u) /\
    (trans_preRequisite e x prq (getId p) prin_u))  $\rightarrow$ 
    (trans_policy_positive e x p prin_u a))

  | PrimitiveExclusivePolicySet prq p  $\Rightarrow$   $\forall$  x, (((trans_prin x prin_u) /\
    (trans_preRequisite e x prq (getId p) prin_u))  $\rightarrow$ 
    (trans_policy_positive e x p prin_u a)) /\
    ((not (trans_prin x prin_u))  $\rightarrow$  (trans_policy_negative e
    x p a)))

  | AndPolicySet ps_list  $\Rightarrow$  trans_ps_list ps_list prin_u a
  end.

```

Translation of a *prin* (called *trans_prin* in listing 7.4) 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.

Listing 6.4: Translation of a Prin

```

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

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

```

A positive translation for a policy (called *trans_policy_positive* in listing 7.5) 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* \rightarrow *act* \rightarrow *asset* \rightarrow *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 *policies*, the conjunction of *trans_policy_positives* are returned.

Listing 6.5: 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  $\Rightarrow$  trans_policy_positive e x p1 prin_u a
  | NewList p p_list'  $\Rightarrow$ 
    ((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  $\Rightarrow$  ((trans_preRequisite e x prq (Single
  policyId) prin_u)  $\rightarrow$ 
    (Permitted x action a))
| AndPolicy p_list  $\Rightarrow$  trans_p_list p_list prin_u a
end.

```


A negative translation for a policy (called *trans_policy_negative* in listing 7.6) 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. Note that the notation (\neg) indicates that *Permitted* may be negated. As 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 *policies*, the conjunction of *trans_policy_negatives* are returned.

Listing 6.6: 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 7.7) 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 7.7 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 (3.12) they will be conjunctions, disjunctions and exclusive disjunctions of translations for each *prerequisite*.

Listing 6.7: Translation of a PreRequisite

```

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

```

```

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

```

The translation of a *constraint* (called *trans_constraint* in listing 7.8) 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*.

In listing 7.8 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*.

Listing 6.8: 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  $\Rightarrow$  trans_prin x prn

| Count n  $\Rightarrow$  trans_count e n IDs prin_u

| CountByPrin prn n  $\Rightarrow$  trans_count e n IDs prn

end.

```

The translation of a *forEachMember* (called *trans_forEachMember* in listing 7.9) 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_constraints* (for the first pair) and *trans_forEachMember_Auxs* (for the rest of the pairs) are returned.

Listing 6.9: 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  $\Rightarrow$  trans_constraint e x (right pair1) IDs (Single (left pair1))
  | NewList pair1 rest_pairs  $\Rightarrow$ 
    (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 7.10) 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 to return the negation of *trans_constraint* (see listing 7.8).

Listing 6.10: Translation of not cons

```

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 7.11) 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 *subjects*, the addition of *get_count* (for the first pair) and *trans_count_auxs* (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).

Listing 6.11: 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 7

Queries

7.1 Introduction

We first mentioned queries in chapter 5 on page 13. 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 [6] 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 [6] which we will use to prove decidability results of our own.

The translation functions plus the auxiliary types and infrastructure, implementing the semantics have been encoded in Coq. Translation functions all build terms of *sort Prop*.

Talk about Props here...

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 ?? for more details on odr0). 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.

Listing 7.1: Environments and Counts

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

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

7.2 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*.

Listing 7.2: 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 7.3), 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_{pslist}* has been defined where for a single *policySet*, *trans_{ps}* is called, and for a list of *policySets*, the conjunction of *trans_{ps}* are returned.

Listing 7.3: 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  $\Rightarrow$  trans_ps e ps1 prin_u a
  | NewList ps ps_list'  $\Rightarrow$  ((trans_ps e ps prin_u a) /\ (trans_ps_list ps_list' prin_u a
  ))
end) in
  match ps with
  | PrimitivePolicySet prq p  $\Rightarrow$   $\forall$  x, (((trans_prin x prin_u) /\
    (trans_preRequisite e x prq (getId p) prin_u))  $\rightarrow$ 
    (trans_policy_positive e x p prin_u a))

  | PrimitiveExclusivePolicySet prq p  $\Rightarrow$   $\forall$  x, (((trans_prin x prin_u) /\
    (trans_preRequisite e x prq (getId p) prin_u))  $\rightarrow$ 
    (trans_policy_positive e x p prin_u a)) /\
    ((not (trans_prin x prin_u))  $\rightarrow$  (trans_policy_negative e
    x p a)))

  | AndPolicySet ps_list  $\Rightarrow$  trans_ps_list ps_list prin_u a
end.

```

Translation of a *prin* (called *trans_prin* in listing 7.4) 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.

Listing 7.4: Translation of a Prin

```

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

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

```

A positive translation for a policy (called *trans_policy_positive* in listing 7.5) 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* \rightarrow *act* \rightarrow *asset* \rightarrow *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 *policies*, the conjunction of *trans_policy_positives* are returned.

Listing 7.5: 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  $\Rightarrow$  trans_policy_positive e x p1 prin_u a
  | NewList p p_list'  $\Rightarrow$ 
    ((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  $\Rightarrow$  ((trans_preRequisite e x prq (Single
  policyId) prin_u)  $\rightarrow$ 
    (Permitted x action a))
| AndPolicy p_list  $\Rightarrow$  trans_p_list p_list prin_u a
end.

```

A negative translation for a policy (called *trans_policy_negative* in listing 7.6) 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. Note that the notation (\neg) indicates that *Permitted* may be negated. As 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 *policies*, the conjunction of *trans_policy_negatives* are returned.

Listing 7.6: 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 7.7) 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 7.7 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 (3.12) they will be conjunctions, disjunctions and exclusive disjunctions of translations for each *prerequisite*.

Listing 7.7: 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 7.8) 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*.

In listing 7.8 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*.

Listing 7.8: 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 7.9) 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_constraints* (for the first pair) and *trans_forEachMember_Auxs* (for the rest of the pairs) are returned.

Listing 7.9: 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 7.10) 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 to return the negation of *trans_constraint* (see listing 7.8).

Listing 7.10: Translation of not cons

```

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 7.11) 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 *subjects*, the addition of *get_count* (for the first pair) and *trans_count_aux* (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 *prin* parameter for *CountByPrin* which allows for getting counts for subjects not necessarily the same as *prin_u*, the agreement's user(s).

Listing 7.11: 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 8

Examples

8.1 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 [6].

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 start with some specific propositions/theorems about these examples to get a feel for how proofs are done in Coq.

8.2 Agreement 2.1

Consider example 2.1 (from [6]) where the *policySet* is a *AndPolicySet* with *p1* and *p2* as the individual *policySets*. 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 8.1: Agreement 2.1 (as used in [6])

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

The Coq version of the agreement 2.1 (listing 8.1) 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 8.1.

Listing 8.2: 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))).

```

8.3 Agreement 2.5

Consider example 2.5 (from [6]) 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 *PrimitivePolicies*. Both policies have prerequisites of type *ForEachMember* with actions *display* and *print* respectively. The *prin* component for both *ForEachMembers* is *Alice, Bob*, whereas the constraint for the first *ForEachMember* is *Count[5]* and for the second is *Count[2]*.

Listing 8.3: Agreement 2.5 (as used in [6])

```

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 8.3) and its sub-parts is listed below. See agreement 2.5 listed in 8.3 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 8.4: 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.

```

8.4 Agreement 2.6

Consider example 2.6 (from [6]) where the *policySet* is a *PrimitiveExclusivePolicySet* with a *InSequence requirement* as prerequisite. We will cover requirements type constraints in *ODRL1* since in *ODRL0* we have elided their use. We will also describe this example in detail when *ODRL1* constructs are added.

Listing 8.5: Example 2.6

```

Definition prins2_6 := prins2_5.

Definition aliceCount10:preRequisite := Constraint (CountByPrin (Single Alice) 10).
Definition primPolicy2_6:policy := PrimitivePolicy aliceCount10 id3 Play.
Definition policySet2_6_modified:= PrimitiveExclusivePolicySet TruePrq
  primPolicy2_6.

```

Chapter 9

Some Simple Theorems

9.1 Introduction

In this chapter we will declare and prove some very simple theorems about the examples from chapter [8]. 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*.

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 any identifier whose type is a proposition ([3]). 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 9.1 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. 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” ([2]).

Listing 9.1: Proof Example

Theorem example1: $\forall x:\text{nat}, x < x + 1$.

9.2 Theorem One

In listing 9.3 we define a *policySet* with a *constraint* such that if *Alice* has used the policy with *id1* 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 *id1* to justify her printing 2 times in an environment called *eA1*. Recall that environments are defined to be non-empty lists of *count_equality* objects (see 7.1).

We also declare a hypothesis *H* with the proposition that results from the translation of the agreement (see definition of *trans_agreement* in listing 7.2) 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 9.2: Hypothesis for Theorem One

$\forall x : \text{subject}, x = \text{Alice} \wedge \text{True} \rightarrow 2 < 5 \rightarrow \text{Permitted } x \text{ Print TheReport}.$

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 9.3: 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.

```

9.3 Theorem Two

In listing 9.5 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 below :

Listing 9.4: 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 9.5: 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.

```

9.4 Theorem Three

In listing 7.1 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 9.6 defines the binary predicate *inconsistent*.

Listing 9.6: 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 9.7. 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 9.7: 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 9.8). 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 9.8: 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 9.9).

Listing 9.9: 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:
   $\forall f g, \text{inconsistent } f g \rightarrow \sim \text{env\_consistent } (\text{ConsEnv } f (\text{SingleEnv } g)).$ 
Proof.
intros. unfold not. intros H'.
inversion H'. intuition. intuition. Qed.

Theorem e2_is_inconsistent:  $\sim \text{env\_consistent } e2.$ 
Proof.
apply two_inconsistent_formulas_imply_env_inconsistent.
apply f1_and_f2_are_inconsistent. Qed.

Theorem env_consistent_implies_two_consistent_formulas:
   $\forall (f g: \text{count\_equality}),$ 
   $\text{env\_consistent } (\text{ConsEnv } f (\text{SingleEnv } g)) \rightarrow \sim \text{inconsistent } f g.$ 
Proof.
intros. inversion H. exact H1. intuition. Qed.

Theorem two_consistent_formulas_imply_env_consistent:
   $\forall (f g: \text{count\_equality}),$ 
   $\sim \text{inconsistent } f g \rightarrow \text{env\_consistent } (\text{ConsEnv } f (\text{SingleEnv } g)).$ 
Proof.
intros. apply consis_2. exact H. Qed.

Theorem env_inconsistent_implies_two_inconsistent_formulas:
   $\forall (f g: \text{count\_equality}),$ 
   $\sim \text{env\_consistent } (\text{ConsEnv } f (\text{SingleEnv } g)) \rightarrow \text{inconsistent } f g.$ 
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 : $\forall (s1\ s2:$
 subject),
 $\forall (id1\ id2: policyId),$
 $\forall (n1\ n2: nat),$

$(s1 = s2 \wedge id1 = id2 \wedge n1 \neq n2) \rightarrow$
 inconsistent (CountEquality s1 id1 n1) (CountEquality s2 id2 n2).

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

Chapter 10

Proposed Future Work

10.1 Introduction

By defining formal semantics for ODRL authors of [6] were able to show some important results. First result is that answering the question of whether a set of ODRL statements imply a permission or a denial (or other possibilities as we will see) is decidable and also that its complexity is NP-hard (see Theorem 4.1 from [6] re-printed here below).

Theorem 4.1 *The problem of deciding, for a query $q = (A, s, act, a, E)$, whether f_q^+ is E-valid is decidable but NP-hard. Similarly for f_q^- .*

The authors of [6] then prove that by removing the construct *not[policySet]* from ODRL's syntax makes answering the same query 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 10.1 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 10.1: Environments and Counts

```
Theorem queriesAreDecidable:  $\forall$  (e:environment),  
     $\forall$  (agr: agreement),  
     $\forall$  (s:subject),  
     $\forall$  (action:act),  
     $\forall$  (a:asset),  
  
    (permissionGranted e [agr] s action a)  $\vee$   $\vee$   
    (permissionDenied e [agr] s action a)  $\vee$   $\vee$   
    (queryInconsistent e [agr] s action a)  $\vee$   $\vee$   
    (permissionUnregulated e [agr] s action a).
```

We will state and prove the complexity (expecting a polynomial time complexity) of the decision procedure (see [1] for a detailed example of an algorithm complexity proof).

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 prove decidability and NP-hardness as stated in theorem 4.1.

10.2 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 eXtensible Access Control Markup Language (XACML) and Security Enhanced Linux (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 and denies the access request [sarna-starosta]

10.3 SELinux Policy Language

The SELinux policy has four different kinds of statements: declarations, rules, constraints and assertions [archer]. Assertions are compile time checks that the *checkpolicy* tool per-

forms at compile time. The other three kinds of statements however are evaluated at run-time.

Declaration statements declare a user, a role and a type.

Listing 10.2: Declarations

```
user u types Ru;

role r types Tr;

type t, attrib_{1}, ..., attrib_{n};
```

Rule statements define access vector rules and type transition rules. Access vector (AV) rules (see 10.3) 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 *sourcetype* are allowed to perform operations in *perm* on objects with class *obj-class* and type *targettype*. *auditallow* means to allow and audit, *dontaudit* means to never audit and finally *neverallow* provides a mechanism to override allow rules. When a process changes security context, the role may change, assuming a role transition rule exists relating the old and the new roles (??).

Listing 10.3: AV rule

```
avkind sourcetype targettype:object-class perm

avkind=allow, auditallow, dontaudit, neverallow
```

Constraints are additional conditions that must hold for an attempted operation to be allowed. Constrains relate all of their arguments (the security contexts) to the server (see 10.4). Whenever a permission is requested on an obj-class, the security server checks that the two security contexts are related by a constrain statement.

Listing 10.4: Constrain rule

```
constrain classes, perms, sourcetype, sourcerole, sourceuser, targettype, targetrole,
        targetuser
```

10.4 Authorization Relation

The authorization relation characterizes the operations allowed by a policy. $auth(C, P, T1, R1, U1, T2, R2, U2)$ holds if the security context defined by the tuple $\langle T1, R1, U1 \rangle$ and $\langle T2, R2, U2 \rangle$ are *consistent* and an entity with type $T1$ has permission P for targets with class C and type $T2$, and the constraint defined by *constrain* holds. A security context

type $\langle T, R, U \rangle$ (type, role and user) is consistent with respect to a given policy if (1) either there exists a *types* such that a role-type relation $role(R, types)$ holds and T is in *types*, or R is *object_r* (note that since the notion of a role is irrelevant for files, all files are assigned the *object_r* role) and T is not a process type, and (2) either there exists *roles* such that $user(U, roles)$ holds and R is in *roles*, or R is *object_r*.

References

- [1] Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors. *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 of *Lecture Notes in Computer Science*. Springer, 2009.
- [2] Yves Bertot. Coq in a Hurry. Technical report, May 2010.
- [3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. <http://www.labri.fr/perso/casteran/CoqArt/index.html>.
- [4] Robby Robson Geoff Collier, Harry Piccariello. A digital rights management ecosystem model for the education community. *DRM Whitepapers: Content Guard*, 2004.
- [5] Robert C. Martin, Dirk Riehle, and Frank Buschmann. *Pattern Languages of Program Design 3*. Addison-Wesley, Reading, Mass., 1998.
- [6] Riccardo Pucella and Vicky Weissman. A formal foundation for ODRL. *CoRR*, abs/cs/0601085, 2006.