Certifying Digital Rights' Expression Languages

by

Bahman Sistany

Ph.D.Thesis Proposal For the Ph.D. degree in
Electrical and Computer Engineering

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

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 [1].

Rights Expression Languages, *RELs*, or more precisely when dealing with digital assets, Digital Rights Expression Languages *DRELs* 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 [2] specify a predicate logic based based language that represents a subset of ODRL.

2.3 what will I do?

2.3.1 Coq

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

2.4 Abstract Syntax

Authors of [2] use abstract syntax instead of XML to express statements in the ODRL language. The abstract syntax used is a more compact representation than XML based language ODRL policies are written in and furthermore it simplifies specifying the semantics as we shall see. As an example here is an agreement written in ODRL and the comparable agreement expressed in the abstract syntax [2].

Listing 2.1: 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 2.1 is shown below using the syntax from [2].

Listing 2.2: agreement for Mary Smith as BNF (as used in [2])

```
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 expressed as 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.

2.4.1 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 not 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 2.12). Note that we have also removed the condition *not*[*policySet*] from ODRL since the authors in [2] 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 2.2 would look like in ODRL0's Coq version.

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))))).
```

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* (*prin*), an *asset* and a *Policy Set* (*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 for the DRM application we will be using throughout.

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 [2]). *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 [2] 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 2.6 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 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
```

2.5 Coq Version

Listing 2.14: Coq version of agreement

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

Listing 2.15: prin

```
Definition prin := nonemptylist subject.
```

Listing 2.16: asset

```
Definition asset := nat.
```

Listing 2.17: subject

```
Definition subject := nat.
```

Listing 2.18: policySet

```
Inductive policySet : Set :=
  | PrimitivePolicySet : preRequisite → policy → policySet
  | PrimitiveExclusivePolicySet : preRequisite → policy  → policySet
  | AndPolicySet : nonemptylist policySet → policySet.
```

Listing 2.19: policy

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

Listing 2.20: act

```
Definition act := nat.
```

Listing 2.21: policyId

```
Definition policyId := nat.
```

Listing 2.22: 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.
```

Listing 2.23: constraint

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

2.6 Semantics

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.

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.

The translation of an *agreement* returns the translation for a *policySet* per $prin_u$, the agreement's user and a, the asset.

Listing 2.24: Agreement Translation

$$[\![agreement\ \ for\ \ prin_u\ \ about\ \ a\ \ with\ \ ps]\!] \triangleq [\![policySet]\!]^{prin_u,a}$$

The translation for a $[\![policySet]\!]^{prin_u,a}$ is described by describing the translation for each type of *policySet*. A *policySet* is either a *PrimitivePolicySet*, a *PrimitiveExclusivePolicySet* or a *AndPolicySet*.

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 2.25: Policy Set Translation : PrimitivePolicySet

$$[\![preRequisite \rightarrow policy]\!]^{e,prin_u,a} \triangleq \forall x\ (([\![prin_u]\!]_x \wedge [\![preRequisite]\!]_x^{e,getId(p),prin_u,a}) \rightarrow [\![policy]\!]_x^{positive,e,prin_u,a})$$

PrimitiveExclusivePolicySet (*preRequisite* $\mapsto$ *policy*) yields the conjunction of two implications. 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 2.26: Policy Set Translation : PrimitiveExclusivePolicySet

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

AndPolicySet translates to conjunctions of the corresponding policy set translations.

Listing 2.27: Policy Set Translation : AndPolicySet

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

The Coq version of the policy set translation function follows (listing 2.28).

Listing 2.28: 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_u* takes as input x, a variable of type *subject* and proceeds based on whether we have a single subject or a list of subjects.

Translation of a single $prin_u$ yields a formula that is true if and only if the subject x is the same as the single subject.

Listing 2.29: Prin Translation : Single subject

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

Translation of a list of subjects is the conjunction of the translations of each subject.

Listing 2.30: Prin Translation : List of subjects

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

Listing 2.31: 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.
```

Depending on whether a permission is granted or not we define two translations for policies. A positive translation and a negatives one.

A positive translation for a policy p takes as input e, the *environment*, x, the *subject*, $prin_u$, 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 act to a, if the $preRequisite$ holds.

Listing 2.32: Positive Policy Translation : Single policy

$$[\![preRequisite \Rightarrow_{policyId} act]\!]_x^{positive,e,prin_u,a} \triangleq ([\![preRequisite]\!]_x^{e,policyId,prin_u}) \Rightarrow Permitted(x, [\![act]\!], a)$$

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

Listing 2.33: Positive Policy Translation : List of policies

$$[\![and[policy_1, ..., policy_m]]\!]^{positive,e,prin_u,a} \triangleq [\![policy_1]\!]^{positive,e,prin_u,a} \wedge \ ... \ \wedge [\![policy_m]\!]^{positive,e,prin_u,a}$$

Listing 2.34: 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 p takes as input e, the *environment*, x, the *subject*, $prin_u$, 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 forbidden to do *act* to a regardless of whether *preRequisite* holds.

Listing 2.35: Negative Policy Translation : Single policy

$$[\![preRequisite \Rightarrow_{policyId} act]\!]_x^{negative,e,prin_u,a} \triangleq$$
$$([\![preRequisite]\!]_x^{e,policyId,prin_u}) \Rightarrow \neg(Permitted(x, [\![act]\!], a))$$

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

Listing 2.36: Negative Policy Translation : List of policies

$$[\![and[policy_1, ..., policy_m]]\!]^{negative,e,a} \triangleq [\![policy_1]\!]^{negative,e,a} \wedge \ldots \wedge [\![policy_m]\!]^{negative,e,a}$$

Listing 2.37: 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* takes as input e, the *environment*, x, the *subject*, $prin_u$, the agreement's user, and IDs, the set of identifiers (of policies implied by the *prerequisite*) 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*.

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

Listing 2.38: Prerequisite Translation : Always True Prerequisite

$$[\![prerequisite :: true]\!] \triangleq \texttt{True}$$

The translation for a *Constraint* is handled by a specialized constraint translation function (coverage of which starts at 2.46.

Listing 2.39: Prerequisite Translation : Constraint

$$[\![prerequisite :: constraint]\!]_x^{[id_1,...,id_m],prin_u} \triangleq [\![constraint]\!]_x^{[id_1,...,id_m],prin_u}$$

The translation for a *ForEachMember* is also is handled by a specialized translation function (covered at 2.51.

Listing 2.40: Prerequisite Translation : ForEachMember

$$[\![prerequisite :: forEachMember]\!]_x^{[subject_1,...,subject_k],[constraint_1,...,constraint_m],[id_1,...,id_n]} \triangleq [\![forEachMember]\!]_x^{[subject_1,...,subject_k],[constraint_1,...,constraint_m],[id_1,...,id_n]}$$

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

Listing 2.41: Prerequisite Translation : Not Constraint

$$[\![not\ \ prerequisite :: constraint]\!]_x^{[id_1,...,id_m],prin_u} \triangleq \neg[\![constraint]\!]_x^{[id_1,...,id_m],prin_u}$$

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

Listing 2.42: Prerequisite Translation : Conjunction

$$[\![and\ \ [preRequisite_1, ..., preRequisite_k]]\!]^{[id_1,...,id_m],prin_u} \triangleq [\![preRequisite_1]\!]^{[id_1,...,id_m],prin_u} \wedge ... \wedge [\![preRequisite_k]\!]^{[id_1,...,id_m],prin_u}$$

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

Listing 2.43: Prerequisite Translation : Inclusive Disjunction

$$[\![or\ \ [preRequisite_1, ..., preRequisite_k]]\!]^{[id_1,...,id_m],prin_u} \triangleq [\![preRequisite_1]\!]^{[id_1,...,id_m],prin_u} \vee ... \vee [\![preRequisite_k]\!]^{[id_1,...,id_m],prin_u}$$

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

Listing 2.44: Prerequisite Translation : Exclusive Disjunction

$$[\![Xor\ \ [preRequisite_1, ..., preRequisite_k]]\!]^{[id_1,...,id_m],prin_u} \triangleq [\![preRequisite_1]\!]^{[id_1,...,id_m],prin_u} \oplus \\ ... \oplus [\![preRequisite_k]\!]^{[id_1,...,id_m],prin_u}$$

The Coq version of the translate function for a $preRequisite$ is given in 2.45. Note that in this listing the translations for $AndPrqs$, $OrPrqs$ and $XorPrqs$ have not yet been done.

Listing 2.45: 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 $constraint$ takes as input e, the $environment$, x, the $subject$ and IDs, the set of identifiers (of policies implied by the $constraint$) 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 a $Principal$ is handled by a specialized translation function (covered at 2.31.

Listing 2.46: Constraint Translation : Principal

$$[\![constraint :: prin]\!]_x^{[subject_1,...,subject_m]} \triangleq [\![prin]\!]_x^{[subject_1,...,subject_m]}$$

The translation for a $Count$ is handled by a specialized translation function (covered at 2.56.

Listing 2.47: Constraint Translation : Count

$$[\![constraint :: count[N]]\!]_x^{[id_1,...,id_m],prin_u} \triangleq [\![count[N]]\!]_x^{[id_1,...,id_m],prin_u}$$

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 2.56.

Listing 2.48: Constraint Translation : Count by Principal

$$[\![constraint :: prin(count[N])]\!]_x^{[subject_1,...,subject_m],[id_1,...,id_n]} \triangleq$$
$$[\![prin(count[N])]\!]_x^{[subject_1,...,subject_m],[id_1,...,id_n]}$$

The Coq version of the translate function for a $constraint$ is given in 2.49.

Listing 2.49: 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 for $forEachMember$ yields a formula that is true, if each subject mentioned in $forEachMember$ is met by each constraint mentioned in $forEachMember$. Note that the agreement's user $prin_u$ maybe overriden by a $forEachConstraint$ constraint.

Listing 2.50: ForEachMember Translation : Count by Principal

$$[\![forEachMember]\!]_x^{[subject_1,...,subject_k],[constraint_1,...,constraint_m],[id_1,...,id_n]} \triangleq$$
$$[\![constraint]\!]_x^{(subject_1,constraint_1),[id_1,...,id_n]} \wedge \ ... \ \wedge [\![constraint]\!]_x^{(subject_1,constraint_m),[id_1,...,id_n]}$$
$$\wedge \ ... \ \wedge [\![constraint]\!]_x^{(subject_2,constraint_1),[id_1,...,id_n]} \wedge \ ... \ \wedge$$
$$[\![constraint]\!]_x^{(subject_2,constraint_m),[id_1,...,id_n]} \wedge \ ... \ \wedge [\![constraint]\!]_x^{(subject_k,constraint_1),[id_1,...,id_n]}$$
$$\wedge \ ... \ \wedge [\![constraint]\!]_x^{(subject_k,constraint_m),[id_1,...,id_n]}$$

Listing 2.51: 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 for "Not Constraint" was listed in listing 2.41 earlier but we repeat it here to go along the Coq version.

Listing 2.52: Not Constraint Translation

$$[\![not\ \ constraint]\!]_x^{[id_1,\ldots,id_m],prin_u} \triangleq \neg[\![constraint]\!]_x^{[id_1,\ldots,id_m],prin_u}$$

Listing 2.53: 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 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 2.54: Count Translation : subject and policyId pair

$$[\![count[N]]\!]_x^{subject_1,id_1} \triangleq getCount(subject_1, id_1) < N$$

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 2.55: Count Translation : subject and policyId pairs

$$[\![count[N]]\!]_x^{[id_1,\ldots,id_m],prn} \triangleq$$

$$(getCount(getSubject(prn)_1, id_1)\ +\ldots+\ getCount(getSubject(prn)_1, id_m)\ +\ldots+\ getCount(getSubject(prn)_k, id_1)\ +\ldots+\ getCount(getSubject(prn)_k, id_m))\ <\ N$$

Listing 2.56: 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.
```

2.7 Semantics in Coq

A predicate in Coq is any function that returns a term of type *Prop*. Here is how we have specified the *Permitted* predicate in Coq: $ParameterPermitted : subject- > act- > asset- > Prop$. So *Permitted* predicate takes a *subject*, an *act* (action) and an *asset* and returns a *Prop*. Note that the notation ($\neg$) indicates that *Permitted* may be negated.

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 2.4.1 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 2.57: 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.
```

Translation of the top level *agreement* element proceeds by case analysis on the structure of the agreement. Note that each translation function takes an environment parameter.

Listing 2.58: 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* takes as input $prin_u$, 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 $AndPolicySet$.

References

[1] Robby Robson Geoff Collier, Harry Piccariello. A digital rights management ecosystem model for the education community. *DRM Whitepapers: Content Guard*, 2004.

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