April 2015

Scenarios for Description Logic

Erica Danielle Ford
Worcester Polytechnic Institute

Nicholas Daniel Murray
Worcester Polytechnic Institute

Scenarios for Description Logic

Project Number: DJD-AANW

A Major Qualifying Project

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Erica Ford

Nicholas Murray

Date: May 2015

APPROVED:

Professor Daniel Dougherty, MQP Advisor

Abstract

Description logics form a family of knowledge representation languages for modeling ontologies. Model-finding is a technique for analyzing a first-order theory T by constructing and querying the models of T. This project develops a translation from description logic theories to first-order theories, enabling the use of the Razor model-finder. We provide an implementation of the translation algorithm, and a proof of correctness of the algorithm. As a case study, we explore a sample role-based access control policy formalized in description logic and show how to reason about using Razor.

Acknowledgements

We would like to thank our advisor, Professor Dougherty, for providing direction and constant help throughout our MQP.

We woud also like to thank Ryan Danas and Salman Saghafi for help learning and troubleshooting Razor.

Finally, we would like to thank all the professors who have taught, assisted, and encouraged us throughout our careers at WPI.

Contents

List of Figures

List of Tables

Code

Chapter 1

Introduction

Model finding is a method for examining sets of logical axioms for consistency and possible consequences. Users can either examine the existence of models generated by a set of logical axioms or delve into individual models to see specific elements. The ability to examine individual models distinguishes model finding from theorem proving: the latter can, of course only allow a user to infer the existence or non-existence of models. Model finding therefore has the potential to give a user richer information than they would get using deductive reasoning tools.

The main contribution of this project was bringing model finding capabilities to Description Logic. Description Logics are a family of languages used to formalize knowledge bases, and are frequently used to encode knowledge about medical and life science fields and the Semantic Web. Tools for reasoning about Description Logic are predominantly consistency and satisfiability checking tools, with a few exceptions noted in Section 3.1.1 [16]. With the increased power of model finders, Description Logic ontology builders should be able to get a richer picture of the logical consequences of their statements and potentially spot unexpected behavior. In order to bring model finding capabilities to the Description Logic community, a translation algorithm from Description Logic to Geometric Form was created and implemented. The translation algorithm was necessary in order to prepare Description Logic ontologies to be read by Razor [15], a minimal model finding tool that requires axioms in Geometric Form. Razor was chosen as the preferred model finding tool because of its querying and provenance functionality (see Section 3.3.2) and as a test case for its capabilities.

In addition to defining the translation algorithm, the algorithm was implemented using Java and its correctness was proven. To demonstrate the efficacy of the algorithm, a case study exploring access control policies using Razor was designed. The case study not only demonstrates the soundness of the translation algorithm, but also furthers work by the Description Logic community exploring the benefits of formalizing access control policies using Description Logic [1] [5]. By exploring access control policies first formalized using Description Logic and then translated into Geometric Form for Razor, the capability of model finding on access control policies was introduced for the first time to the Description Logic community.

This paper is organized as follows: In chapter 2, an introduction to first order logic, geometric form, and description logic is presented; In chapter 3, current research in the Description Logic and model finding communities is discussed; In chapter 4, the translation algorithm from Description Logic to Geometric Form is introduced alongside its proof of correctness; In chapter 5, the aforementioned case study involving access control policies explored in Razor is explored.

Chapter 2

Background

2.1 Introduction to First Order Logic

Before beginning an introduction to Description Logic, it is important to understand first order logic. First Order Logic is a formal system with the following definitions.

2.1.1 Syntax

Definition 1 (Vocabulary)**.** A *vocabulary* consists of

- a set of relation symbols, each with a specified natural number arity.
- a set of constant symbols.
- a set of function symbols

Definition 2 (Terms)**.** Fix a countably infinite sequence $Vars = v_1, v_2, ...$ of *variables.* Fix a vocabulary $\mathcal{L}$. The set of *terms* over $\mathcal{L}$ is given inductively by

- a *variable* is a term;
- if f is a function symbol with arity k and $t_1, ..., t_k$ are each terms, then $f(t_1, ..., t_k)$ is a term.

Functions of arity 0 are called *constant symbols* and are usually written as c, for example, rather than $c()$, and are still considered terms.

Definition 3 (Formula)**.** Fix a vocabulary $\mathcal{L}$. The set of *formulas* over $\mathcal{L}$ is given inductively by

1. whenever R is a relation symbol of arity k and each of $t_1, ..., t_k$ is a term, then $R(t_1, ..., t_k)$ is a formula, known as an *atomic formula.*
2. If t_1 and t_2 are terms, then $t_1 = t_2$ is a formula. This is also known as an *atomic formula.*

3. If α is a formula, then $\neg\alpha$ is a formula.
4. If α and β are formulas, then $(\alpha \wedge \beta)$ is a formula.
5. If α and β are formulas, then $(\alpha \vee \beta)$ is a formula.
6. If α and β are formulas, then $(\alpha \Rightarrow \beta)$ is a formula.
7. If α is a formula and x is a variable, then $(\forall x.\alpha)$ is a formula.
8. If α is a formula and x is a variable, then $(\exists x.\alpha)$ is a formula.

Definition 4 (Free and Bound Variables)**.** The set of *free variable occurrences* in a formula α is defined inductively as:

- Any variable occurrence in an atomic formula is a free occurrence.
- The free variable occurrences in $\neg\alpha$ are the free variable occurrences in α.
- The free variable occurrences in $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, and $(\alpha \Rightarrow \beta)$ are the free variable occurrences in α together with the free variable occurrences in β.
- The free variable occurrences in $(\forall x.\alpha)$ are the free variable occurrences in α, *other than* the occurrences of x in α. These occurrences of x in α are the bound variable occurrences.
- The free variable occurrences in $(\exists x.\alpha)$ are the free variable occurrences in α, *other than* the occurrences of x in α. These occurrences of x in α are the bound variable occurrences.

Definition 5 (Semantics of Formulas)**.** Fix a signature $\mathcal{L}$. Let $\mathbb{M}$ be a $\mathcal{L}$-model and let α be a formula. Given environment η for $\mathbb{M}$ we define the notion

$$\mathbb{M} \models \alpha \tag{2.1}$$

by the following induction:

- $\mathbb{M} \models_\eta R(t_1, ..., t_k)$ if the tuple $\{a_1, ..., a_k\}$ is in the relation $R^{\mathbb{M}}$, where for each i, a_i is the element $\eta(x)$ if t_i is the variable x, and is $c^{\mathbb{M}}$ if t_i is the constant c.
- $\mathbb{M} \models_\eta t_1 = t_2$ if the tuple a_1 and a_2 are the same element of $|\mathbb{M}|$, where for each i, a_i is the element $\eta(x)$ if t_i is the variable x, and is $c^{\mathbb{M}}$ if t_i is the constant c.
- $\mathbb{M} \models_\eta (\neg\alpha)$ if $\mathbb{M} \models_\eta \alpha$ fails.
- $\mathbb{M} \models_\eta (\alpha \wedge \beta)$ if $\mathbb{M} \models_\eta \alpha$ and $\mathbb{M} \models_\eta \beta$.
- $\mathbb{M} \models_\eta (\alpha \vee \beta)$ if $\mathbb{M} \models_\eta \alpha$ or $\mathbb{M} \models_\eta \beta$.

- $\mathbb{M} \models_\eta (\alpha \rightarrow \beta)$ if $\mathbb{M} \models_\eta \alpha$ fails or $\mathbb{M} \models_\eta \beta$ holds.
- $\mathbb{M} \models_\eta (\forall x.\alpha)$ if for every $a \in |\mathbb{M}|$ we have $\mathbb{M}_{\eta[x \mapsto a]}\alpha$.
- $\mathbb{M} \models_\eta (\exists x.\alpha)$ if for some $a \in |\mathbb{M}|$ we have $\mathbb{M}_{\eta[x \mapsto a]}\alpha$.

Definition 6. ϕ is a *Conjunctive Normal Form* if it is written as: $\forall \vec{x}, \vec{y}, \vec{z}(x_1 \vee x_2 \vee ... \vee x_i) \wedge (y_1 \vee y_2 \vee ... \vee y_j) \wedge ... \wedge (z_1 \vee z_2 \vee ... \vee z_k)$ where $\vec{x}$, $\vec{y}$, and $\vec{z}$ are all literals.

Lemma 7. *For any ϕ in first order logic, there exists ϕ' in Conjunctive Normal Form such that $\models \phi \Leftrightarrow \phi'$.*

Proof. A proof is given on page 25 of [18]. ///

2.1.2 Semantics

Definition 8 (Equisatisfiable)**.** Two formulas α and β are *equisatisfiable* when α has a model if and only if β has a model.

Definition 9 (Semantic Entailment)**.** If $\mathbb{M}$ is a model and α is a formula, then $\mathbb{M} \models \alpha$ if and only if $\mathbb{M}$ satisfies α.

Definition 10 (Model)**.** If $\mathcal{L}$ is a vocabulary, a *model* $\mathbb{M}$ for $\mathcal{L}$ consists of:

- a non-empty set $|\mathbb{M}|$, called the *universe* of $\mathbb{M}$,
- for each relation symbol R of $\mathcal{L}$ with arity k, a relation $R^{\mathbb{M}} \subseteq |\mathbb{M}|^k$,
- for each function symbol f of $\mathcal{L}$ with arity k, a relation $f^{\mathbb{M}} : |\mathbb{M}|^k \rightarrow |\mathbb{M}|$.

Definition 11 (Validity and Satisfiability)**.** Let σ be a sentence and let Σ be a set of sentences. Then:

- The sentence σ is *valid* if for every model $\mathbb{M}$ we have that $\mathbb{M} \models \sigma$.
- The set Σ is *satisfiable* if there is a model $\mathbb{M}$ such that $\mathbb{M} \models \Sigma$.
- We say that Σ *logically implies* σ, also known as Σ *entails* σ, if for every model $\mathbb{M}$ such that $\mathbb{M} \models \Sigma$ we also have $\mathbb{M} \models \sigma$.
- We can write $\Sigma \models \sigma$ as shorthand for Σ logically implies σ.

2.1.3 Skolem Functions

An important tool for working with sets of first order logic theories is the concept of Skolem functions. Skolem functions are named after their creator, Thoralf Skolem, who proved many important results in set and lattice theory. Skolem functions provide the essential tools needed to enable the provenance functionality in Razor, which sets it apart from many other existing model finders. For a discussion about provenance, see Section 3.3.2.

Informally, Skolem functions remove existential quantifiers from formulas by translating the formulas into an equisatisfiable (Section 2.2) form that is quantifier-free. To remove existential quantifiers, we define a new function that relates the existentially-quantified variable to the universally-quantified variable that generated it. For example, consider the following formula:

$$\forall x \exists y P(x, y) \tag{2.2}$$

We may Skolemize the above formula introducing a Skolem function f relating y to x, to get

$$\forall x P(x, f(x)) \tag{2.3}$$

The above formula is said to be in Skolemized form, because all existential quantifiers have been replaced with Skolem functions. A formal definition of Skolem functions is provided below:

Definition 12 (Skolem Functions)**.** Let $\vec{x} = x_1, ..., x_k$ and $\vec{y} = y_1, ..., y_p$. Let $\alpha = \forall \vec{x} \exists \vec{y} \phi(\vec{x}, \vec{y})$ be a formula. We can introduce a set of function symbol $f_1(\vec{x}), ..., f_p(\vec{x})$, called *Skolem functions*, such that $\alpha = \forall \vec{x} \phi(\vec{x}, \{f_1(\vec{x}), ..., f_p(\vec{x})\})$ holds. When α is written as $\alpha = \forall \vec{x} \phi(\vec{x}, \{f_1(\vec{x}), ..., f_p(\vec{x})\})$, α is said to be in *Skolemized form.*

Using the notion of equisatisfiability, we may state the following theorem:

Theorem 13. *Suppose $\mathcal{L}$ is a language and $\alpha \in \mathcal{L}$ is a formula containing free variables $\vec{x} = x_1, ..., x_k$ and existentially quantified variables $\vec{y} = y_1, ..., y_p$. Then the Skolemized formula α' containing free variables $x_1, ..., x_k$ and bound Skolem functions $f_1(x_1), ..., f_k(x_k)$ and α are equisatisfiable.*

That is, the translated formula that uses Skolem functions will be satisfiable if and only if the original formula was satisfiable.

Proof. Let $\vec{x} = x_1, ..., x_k$ and $\vec{y} = y_1, ..., y_p$ be the set of free and bound variables, respectively. Suppose $\alpha = \forall \vec{x} \exists \vec{y} \phi(\vec{x}, \vec{y})$ and $\alpha' = \forall \vec{x} \phi(\vec{x}, \{f_1(\vec{x}), f_2(\vec{x}), ..., f_p(\vec{x})\})$.

First, suppose $\mathbb{M}$ is a model for $\mathcal{L}$ and $\mathbb{M} \models \alpha$. Then define $\mathbb{M}'$ by expanding $\mathbb{M}$ with Skolem functions $f_1(\vec{x}), ..., f_p(\vec{x})$. We define $f_i(\vec{x})$ as the element $y_j \in \mathbb{M}$ that witnesses the truth of α in $\mathbb{M}$. We can then rewrite α as the Skolemized function α'. Therefore, the expanded model $\mathbb{M}' \models \alpha'$.

Next, suppose the model $\mathbb{N} \models \alpha'$. Define $\mathbb{N}^-$ to be the restriction of $\mathbb{N}$ to $\mathcal{L}$. By restricting the model to $\mathcal{L}$, we replace each witnessing Skolem function $f_i(\vec{x})$ with the original element $y_j \in \mathbb{N}^-$ that causes α to hold. Therefore, $\mathbb{N}^- \models \alpha$. ///

The proof shows that α is a theory if and only if the Skolemized theory α' is a theory.

Skolem functions are vital to Razor's provenance feature. For more on Razor's provenance feature, see Section 3.3.2.

2.2 Geometric Form

Definition 14 (Positive Existential Formula)**.** Positive existential formulas are defined as in Definition 3, but positive existential formulas only permit clauses

1, 2, 4, 5, 6, and 8 as a valid formulas.

Definition 15 (Geometric Form)**.** If sentences α and β are positive existential formulas as defined in Definition 14, then a theory T is in *Geometric Form* if all sentences are written in the form $\alpha \Rightarrow \beta$.

The semantics of geometric form are equivalent to those of first order logic, defined in Section 2.1.2.

Geometric form is a fragment of first order logic, and restricts the format of formulas to those in positive-existential form. Naturally, several questions arise about the expressive power of geometric form and the translation of axioms relative to first order logic. The theorem stated below shows that we do not lose expressive power when we translate formulas from first order logic into geometric form.

Theorem 16. *Geometric form and first order logic are* equisatisfiable*.*

Proof. Let $\alpha = \forall \vec{x} \exists \vec{y} \phi(\vec{x}, \vec{y})$ be a formula in first order logic as defined in Definition 3. We can convert α into the Skolemized formula $\alpha' = \forall \vec{x} \phi(\vec{x}, f(\vec{x}))$, which is equisatisfiable with α, according to Theorem 13. Then, we may convert α' to conjunctive normal form (See Definition 6.) according to the inductive algorithm given in Theorem 2.3.9 in [18]. Since any formula that is in Conjunctive Normal Form is in Geometric Form, α'' is in geometric form. Therefore any first order logic formula α is equisatisfiable with a formula α'' written in a positive existential form. ///

Thus, in some senses, geometric form is as strong as first order logic when it comes to model finding. Therefore, if we model an ontology using both geometric and first order logic, it is sufficient to check if the geometric form set of theories produces a model in order to find out if the theories are satisfiable. This fact stems from the algorithm that translates a set of first order logic theories to a set of geometric form theories. The translation algorithm preserves the property of satisfiability during the translation; that is, if the first order logic theories were unsatisfiable (satisfiable), the geometric form theories will also be unsatisfiable (satisfiable). Significantly, the translation algorithm does not need to know if the first order logic theories are satisfiable before doing the translation. The translation algorithm discussed in Chapter 4 translates from first order logic to geometric form, and a proof of its preservation of models can be found in Section 4.3.6.

2.3 Description Logic

Description Logic (DL) lies within first order logic, and it enjoys the property that every currently known fragment of Description Logic is decidable.

2.3.1 Syntax

Definition 17 (Concept)**.** A concept C is a unary predicate.

Definition 18 (Role)**.** A role R is a binary predicate between 2 (not necessarily atomic) concepts.

Definition 19 (Formulas)**.** Let Δ be a domain. The set of *DL formulas* over Δ is given inductively by:

1. An *atomic concept* $\mathrm{C} \in \Delta$ is a DL formula.
2. An *atomic role* $\mathrm{R} \in \Delta \times \Delta$ is a DL formula.
3. The universal truth symbol $\top$ is a DL formula.
4. The universal falsehood symbol $\bot$ is a DL formula.
5. If C is an atomic concept, then $\neg\mathrm{C}$ (the *atomic negation* of C) is a DL formula.
6. If C and D are concepts, then $\mathrm{C} \sqcap \mathrm{D}$, known as the *concept conjunction* of C and D, is a DL formula.
7. If R is a role and C is a concept, a *value restriction* $\forall R.C$ is a DL formula.
8. If R is a role, then the *limited existential quantification* $\exists R.\top$ is a DL formula.
9. If C, D are concepts, then $\mathrm{C} \sqcup \mathrm{D}$ (the *concept disjunction* of C and D) is a DL formula.
10. If R is a role and C is a concept, then the *full existential quantification* $\exists R.C$ is a DL formula.
11. If R is a role and $\mathrm{n} \in \mathbb{N}$, then a *numerical restriction* on R, written as $\geq \mathrm{n}\ \mathrm{R}$ or $\leq \mathrm{n}\ \mathrm{R}$, is a DL formula.

12. If C is a concept, then ¬C, known as the *full negation* of C is a DL formula.

13. A *nominal* is a DL formula.

14. If R is a role, then R^-, known as an *inverse property* is a DL formula.

15. If R is a role, C is a concept, and n ∈ ℕ, then a *qualified cardinality restriction* on R, written as ≥ n R.C or ≤ n R.C is a DL formula.

2.3.2 Semantics

Description logic is a fragment of first order logic with some syntactic and semantic differences.

The following definition of Description Logic semantics corresponds with the syntax definition given in Definition 19.

Definition 20 (Description Logic Formula Semantics)**.** Suppose Δ is a domain. We can then define:

1. An *atomic concept* $C \in \Delta$ is a unary predicate describing a group of individuals in the domain which share some common trait.

2. An *atomic role* $R \in \Delta \times \Delta$ is a binary predicate describes a relationship between two members of a concept.

3. The universal truth symbol ⊤ is interpreted as the entire domain Δ.

4. The universal falsehood symbol ⊥ is interpreted as ∅.

5. If C is an atomic concept, then ¬C is interpreted as Δ\ C.

6. If C and D are concepts, then (C ⊓ D) is interpreted as C ∩ D.

7. If R is a role and C is a concept, then a *value restriction* $\forall R.C$ is interpreted as $\{a \in \Delta^I \mid \forall b.(a,b) \in R^I \Rightarrow b \in C^I\}$.

8. If R is a role, then the *limited existential quantification* $\exists R.\top$ is interpreted as $\{a \in \Delta^I \mid \exists b.(a,b) \in R^I\}$.

9. If C and D are concepts, then (C ⊔ D) is interpreted as C ∪ D.

10. If R is a role and C si concept, then the *full existential quantification* $\exists R.C$ is interpreted as $\{a \in \Delta^I \mid \exists b.(a,b) \in R^I \wedge b \in C\}$.

11. If R is a role and n ∈ ℕ, then a *numerical restriction* on R is defined as either:
    - an at-least restriction: $\geq n\ R = \{a \in \Delta \mid |\{b \mid (a,b) \in R\}| \geq n\}$
    - an at-most restriction: $\leq n\ R = \{a \in \Delta \mid |\{b \mid (a,b) \in R\}| \leq n\}$.

12. If C is a concept, then ¬C is interpreted as $\Delta\setminus$ C.

13. A *nominal* is interpreted as {c ∈ Δ | |c| = 1}.

14. If R is a role, then an *inverse property* R^- is interpreted as {(b,a) ∈ R^- | (a,b) ∈ R^I }.

15. If R is a role, C is a concept, and n ∈ ℕ, then a *qualified cardinality restriction* on R is interpreted as either:

    - an at-least restriction: ≥ n R.C = {a ∈ Δ | |{b | (a,b) ∈ R ∧ b ∈ C }| ≥ n}
    - an at-most restriction: ≤ n R.C = {a ∈ Δ | |{b | (a,b) ∈ R ∧ b ∈ C }| ≤ n}.

Definition 21. An *interpretation* function I over the domain of interpretation Δ^I maps every atomic concept C to $C^I \subseteq \Delta^I$ and every atomic role R to $R^I \subseteq \Delta^I \times \Delta^I$.

The following definitions describe a way of extending the expressibility of a DL fragment by adding transitive closures to roles.

Definition 22 (Transitive Closure)**.** Suppose R is a binary relation, $R \subseteq \Delta \times \Delta$. Then the *transitive closure* of R is the smallest relation R^* such that

- $R \subseteq R^*$ and
- $\forall x, y, z \in \Delta$ if $R^*(x, y)$ and $R^*(y, z)$ then $R^*(x, z)$

Definition 23 (Transitive Roles)**.** A Description Logic fragment is said to have *transitive roles* if there is a syntactic construct which, for any role R, designates the role R^* which is the transitive closure of R. Transitive roles are normally seen alongside the $\mathcal{ALC}$ fragment of Description Logic, and this combination is denoted by the letter $\mathcal{S}$.

2.3.3 Fragments of Description Logic

Description Logic is broken into different fragments, each with their own rules and expressive power. A table of the most common pieces that make up fragments is provided in Table 2.1. A more in-depth look at several common fragments is given below.

Fragment Abbreviation	Definition
$\mathcal{C}$	Full Negation
$\mathcal{U}$	Concept Disjunction
$\mathcal{E}$	Full Existential Quantification
$\mathcal{N}$	Numerical Restriction
$\mathcal{O}$	Nominals
$\mathcal{I}$	Inverse Properties
$\mathcal{Q}$	Qualified Cardinality Restriction
$\mathcal{S}$	$\mathcal{ALC}$ + Transitive Roles
$\mathcal{H}$	Role Hierarchy
$(\mathcal{D})$	Datatype Properties
$\mathcal{R}$	Complex Role Axioms

Table 2.1: Common Description Logic Fragments

The $\mathcal{AL}$-Family of Languages

The $\mathcal{AL}$ family of languages is a set of attributive languages which give a specific syntax and semantics for making logical statements. $\mathcal{AL}$ is the basis for many fragments of Description Logic, and logical components can be added and subtracted to make more or less expressive logics. $\mathcal{AL}$ has a base set of logical assertions which are allowed in statements. The allowable set of logical axioms for $\mathcal{AL}$ correspond to clauses 1, 3, 4, 5, 6, 7 and 8 as defined in Definition 19.

It is possible to disallow some of these base logical assertions to create different fragments of logic. For example, the $\mathcal{FL}^-$ language falls within the $\mathcal{AL}$ family of languages, and it allows all of the base logical concepts with the exception of atomic negation (clause 5). Additionally, the $\mathcal{FL}_0$ language is a further subset of the $\mathcal{AL}$ logic; it is a fragment of $\mathcal{FL}^-$ that also disallows limited existential quantification (Definition 8).

It is also possible to make the $\mathcal{AL}$ family of languages more expressive by adding additional possible logical expressions. Standard notation in Description Logic is to represent each new logical concept with a letter; these letters are then concatenated to the $\mathcal{AL}$ abbreviation as shorthand for the complete range of logical expressions that are allowed. The most common logical concepts that are abbreviated to a single letter are listed in Table 2.1. The language $\mathcal{ALC}$, for example, contains the base set of logical expressions allowed in $\mathcal{AL}$, as well as the power of full negation, denoted by $\mathcal{C}$ (Definition 12). Importantly, the addition of possible logical concepts can make a logic intractable. Increased expressibility within a logic can come at the cost of efficiency.

OWL

The Web Ontology Language (OWL) is a subset of Description Logic that is most frequently used to formalize ontologies and taxonomies. OWL can be expressed as a subset of the $\mathcal{AL}$ family of languages. More specifically, it is classified as $\mathcal{SHOIN}(\mathcal{D})$. The letters of the $\mathcal{SHOIN}(\mathcal{D})$ abbreviation are listed in Table 2.1.

The OWL framework actually consists of 3 fragments of logic which vary in expressibility: OWL LITE, OWL DL, and OWL FULL. These fragments are not completely independent, and the distinctions between the three languages are not clear cut. For example, two ontologies written in OWL LITE may combine to form an OWL FULL ontology. The main differences between the three fragments lie in their tractability and expressive power. OWL LITE is the least expressive of all three, but offers a faster implementation. OWL DL is the most commonly used fragment, and it balances being expressive enough to support rich ontologies while still being practical to implement. While OWL FULL would give the full implementation of $\mathcal{SHOIN}(\mathcal{D})$, no group has actually provided a working implementation of OWL FULL, since it is undecidable [19]. Thus, one of OWL's main shortcomings is the hazy boundary between its three fragments, which causes ontologies to unexpectedly move from practical to reason over into an intractable space. A full review of the shortcomings of OWL is discussed in [7].

OWL2.0

OWL2.0 [14] is the successor to OWL and was created by a W3C working group to address the shortcomings of OWL. OWL had several issues not only as a semantic framework, but also as a standard of logic. Thus, OWL2.0 was created as a separate fragment of Description Logic and is best represented as $\mathcal{SROIQ}$. The abbreviation for $\mathcal{SROIQ}$ is defined in Table 2.1. The advantages of OWL2.0 over OWL are discussed in [7].

2.3.4 Fragment Complexity

Analyzing the complexity of Description Logic fragments can be rather challenging. As syntax restrictions are added and removed, the complexity of the fragments can change dramatically. Evgeny Zolin put together a tool [20] in order to compile as much knowledge about fragment complexity as possible. Results about the most common Description Logic fragments are summarized from [20] below unless otherwise noted.

$\mathcal{ALC}$

Checking the consistency and satisfiability of any set of theories in $\mathcal{ALC}$ is polynomial time complete, according to [17].

OWL

There are three main subsets of OWL: OWL-Lite, OWL-DL, and OWL-Full, as described in 2.3.3. Each fragment has its own complexity.

OWL-LITE is the fragment $\mathcal{SHIF}$, and is Exp time complete.

OWL-DL is the fragment $\mathcal{SHOIN}$ and is the most commonly used fragment of OWL. Checking the consistency and satisfiability of any set of theories in $\mathcal{SHOIN}$ is NExp time complete according to [13]. OWL-DL is decidable according to [19].

OWL-FULL is the fragment $\mathcal{SHOIN}(\mathcal{D})$. Checking the consistency and satisfiability of any set of theories in $\mathcal{SHOIN}(\mathcal{D})$ is NExp time complete according to [13]. Unfortunately, OWL-FULL is undecidable, as shown in [19].

Importantly, infinite models can be created using $\mathcal{SHIF}, \mathcal{SHOIN}$, and $\mathcal{SHOIN}(\mathcal{D})$, which may make reasoning over OWL-LITE ontologies more difficult [10].

OWL2.0

OWL2.0 is the fragment $\mathcal{SROIQ}$. Checking the consistency and satisfiability of any set theories in $\mathcal{SROIQ}$ is NExp time hard and decidable, as shown in [9]. Once again, however, we may encounter infinite models, as shown in [10].

Other Fragments

Here is a listing of some other fragments of Description Logic between$\mathcal{ALC}$ and $\mathcal{SROIQ}$ and their complexities, as accoring to [20].

Fragment	Satisfiability
$\mathcal{ALCF}$	PSpace Complete
$\mathcal{ALCN}$	PSpace Complete
$\mathcal{ALCQ}$	PSpace Complete
$\mathcal{ALCO}$	PSpace Complete
$\mathcal{ALCOQ}$	PSpace Complete
$\mathcal{SH}$	ExpTime Complete
$\mathcal{SHO}$	ExpTime Complete
$\mathcal{SHI}$	ExpTime Complete
$\mathcal{SHN}$	ExpTime Complete
$\mathcal{SHOIN}$ (OWL)	ExpTime Complete
$\mathcal{SHROIN}$	NExpTime Hard

Table 2.2: Satisfiability Complexity for some DL Fragments

Chapter 3

Current Theory and Research

3.1 Current Theory and Research

3.1.1 Related Work

There is only one existing model finder that accepts Description Logic as input. Only SuperModel, a prototype developed by Johannes Bauer at the University of Manchester for his thesis, accepts Description Logic ontologies and presents users with models to explore [3].

In [3], Bauer states that there is a dearth of model exploration tools (and an even greater lack of model augmentation tools), since a great deal of research has instead been focused on axiom-inspection tools to assist ontology builders. Additionally, he proposes that the most useful way to present users with models is to present a small, finite model that users can then augment with further information (See page 23 of [3].). This strategy fits nicely with Razor's generation of minimal models which can be augmented during exploration by the user. Futhermore, Bauer found promising results indicating that users may be very welcome to the development of new model exploration tools. He specifically points out that presenting users with the provenance information about model elements may be particularly interesting to users (See page 66 of [3].).

In [4], Bauer, Sattler, and Parsia have investigated the usefulness of model exploration tools for understanding ontologies. They specifically use SuperModel in a case study to examine whether users found it helpful in reasoning over a small ontology. The authors concluded that users who were familiar with the graphical nature of ontology editors derived the most benefit from SuperModel's ability to present and allow augmentation of models. These findings indicate a niche that Razor may be able to fill, since it gives users the power to explore, augment, and gather provenance information from succinct, minimal models.

3.2 Model Finders

3.2.1 Origins of Model Finding

The concept of exploring the existence of models began with the development of tools to check the consistency and satisfiability of sets of axioms. Software known as SAT-Solvers check the satisfiability of a given set of sentences. While SAT-Solver are very useful for finding flaws in logical theorems, they typically do not provide a model of scenarios in which the theorems hold. Model finders such as Razor do more than just check for satisfiability, by displaying situations to the user that are entailed by the theorems. This practice is highly useful, since models may present scenarios that are logically sound but display an unintended consequence of the set of axioms. In situations such as security policy modeling and hospital privacy laws, seeing the unexpected scenarios of certain guidelines may allow the designers to prevent security or privacy breaches before they happen.

3.2.2 Alloy

Alloy [12], built by Daniel Jackson at MIT, finds models that satisfy given constraints and shows the user one at a time on command. Alloy also displays a graphical representation of the model, which aids users in visualizing the semantic web of information defined in the model. Alloy is a well-known model finding tool, and textbooks such as *Software Abstractions* by Daniel Jackson give thorough overviews of modeling knowledge in Alloy [11]. While Alloy's graphical user interface is a major advantage to users, Alloy does not necessarily present minimal models for each input. Thus, users may find that it is difficult to predict what model Alloy will present. Additionally, Alloy uses its own language as input syntax [12]. Additionally, users looking to employ Alloy as a model finder can also write axioms in first order logic in addition to writing ontologies in Alloy's language.

3.2.3 Razor

Razor [15] is a model finder developed by Daniel Dougherty at Worcester Polytechnic Institute. An important feature of Razor is that it finds a minimal model for given requirements. That is, no elements will be presented in the model that are not required by at least one axiom. While finding the minimal model of a set of axioms can be difficult, the resulting model is very useful because it can potentially provided provenance information 3.3.2. Razor is also unique in that it uses geometric form as its input logic.

3.3 Our Work and Research

3.3.1 Advantages of Geometric Form

There are several key advantages to using Geometric Form in a model finder. First, Geometric Form allows for the verification of the hypothesis of any sentence using a finite subset of (a possibly infinitely large set of) information. The hypothesis of any sentence in positive existential form may contain only the operators $\exists, \vee$, and $\wedge$. Checking an existential quantifier requires searching for the existence of an element that satisfies the hypothesis, or, failing that, the creation of a new such element. Checking the satisfiability of $\alpha \vee \beta$ requires only a search for a case when either α or β holds. Lastly, checking the satisfiability of $\alpha \wedge \beta$ can be halted when a case where either α or β fail to hold.

Additionally, Geometric Form allows for the incremental building of models. Due to the absence of negation, a model finder can incrementally build a model without later introducing a contradiction. For example, if a model finder requires some α to be true, it will never later need to require $\neg\alpha$ to hold.

3.3.2 Razor and Provenance

One useful feature of Razor is provenance, a word derived from the French word provenir, which means "to come from". It historically was used in reference to art and describes the path that an given art piece takes as it travels from owner to owner [2]. Razor's use of provenance, however, refers to the idea of tracking why a model output by Razor has a certain feature. After running Razor on an input file containing Geometric Form theories, Razor will produce a minimal model for the user to explore; users have the option of querying Razor about from where a specific element in the model came. The provenance function was implemented in Razor in the hopes of giving users a clear idea of why individual elements appear in their models; with this information, users can then edit the specific theory that generated said element.

Razor keeps track of element provenance using Skolem functions (See 2.1.3 for a definition of Skolem functions). In particular, since Razor only constructs minimal models, every element in a model is denoted by a closed term built from Skolem functions. Furthermore, since every Skolem function is associated with a particular existential quantifier in the user's input theory, Razor can records these associations, and can map each model element to a formula occurrence (and associated variable binding) in the theory. When a user queries Razor to find where an element of the model came from, Razor can look up this binding and presents the axiom to the user that corresponds to that Skolem function.

3.3.3 Model Exploration and Augmentation

Razor can also explore different models and augment them with new constraints given by the user. Like SuperModel, Razor allows users to explore models to better understand the knowledge base and logical assertions. Provenance also allows a user to pick an element of the model and ask to see which axiom generated that element. The combination of model augmentation, the ability to explore different minimal models, and provenance gives users greater power to reason over the domain of interest.

Chapter 4

Translations

Our translation algorithm implementation can take a two different input types. The first input type is the OWL XML file. The second is a file using a self made framework that allows us to write Description Logic in the a simple ASCII form. We default to taking our Description Logic format and allow OWL as an option. OWL files are translated using a JENA parser to convert them into our Description Logic format. The normal Description Logic translation is then run from that point.

Internally, we copy the data or the ANTLR abstract syntax tree so that we can translate it ourselves as the ANTLR trees allow for only limited manipulations. The tree goes through several different small translations to bring it from DL to GF.

4.1 Parsing

Description Logic files are translated using an ANTLR parser. This parser is generated from a context free grammer, specified below:

Code 4.1: Description Logic Context Free Grammer

```
//Let a file be interpreted as a series of lines
file:  line+ ;

//A line can be an expression, a comment, or just blank
line:   expr1 newline
   |    comment newline
   |    newline
   ;

//Rules for concepts
//Top: DefinedAs and Concept Subsumption
expr1: expr2 SUBSUMPTION expr2 # Subsumption
   |   expr2 DEFINEDAS expr2  # Definedas
   |    expr2                  # Skip1
   ;


//Next: Or
expr2: expr3 OR expr2 # Or
   |      expr3          # Skip2
   ;

//Next: And
expr3: expr4 AND expr3 # And
   |      expr4           #Skip3
   ;

//Next: Not
expr4: NOT expr5 # Not
   |      expr5     # Skip4
   ;


//Next: Quantifiers
expr5:  EXISTS bexpr '.' expr6 # Exists
   |      FORALL bexpr '.' expr6 # Forall
   |      expr6                  # Skip5
   ;

//Last: Names
expr6:  ID         # Name
   |    '(' expr1 ')'  # Parens
   ;

//Role
bexpr: ID #bName
   ;
```

4.2 OWL to GF Algorithm

Most ontologies in the Tones ontology repository only have three different statements despite OWL's level of complexity. They are as follows:

- A is type B
- A is subclassOf B
- A is disjointWith B

Where A and B are atomic.

The translation for these statements is:

Input	Output
A is type B	A(x) $\Rightarrow$ B(x) $\exists$ x. A(x)
A is subclassOf B	A(x) $\Rightarrow$ B(x)
A is disjointWith B	A(x) $\wedge$ B(x) $\Rightarrow \perp$

Table 4.1: OWL to GL Algorithm

Internally, OWL syntax is translated into DL syntax and then input into the DL translator, however, the existing cases are so simple that is can be written as a single direct translation here.

4.3 Description Logic to Geometric Form Algorithm

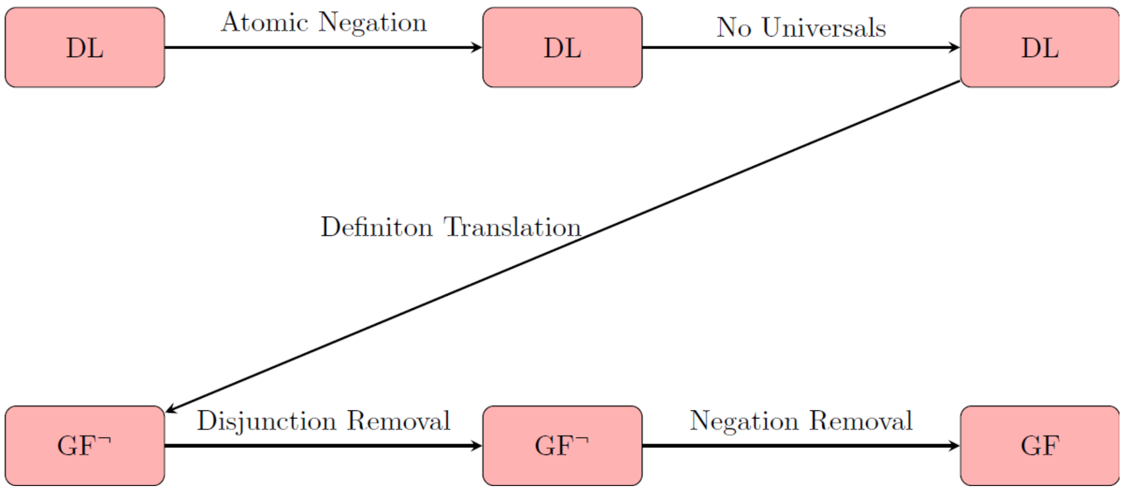


Figure 4.1: Flow Chart of the DL to GF Translation Algorithm

The following is the steps taken by the algorithm to translate Description Logic into the Geometric Form.

4.3.1 Atomic Negation

All negation is pushed down the tree such that only concepts or roles can be negated.

This step is started by applying the NegationSearch algorithm to the top of a tree, it will delve into the tree and upon finding a negation will switch to using the Negate algorithm and correct all issues with negation.

Input	Output
A	A
A $\wedge$ B	NegationSearch(A) $\wedge$ NegationSearch(B)
A $\vee$ B	NegationSearch(A) $\vee$ NegationSearch(B)
$\neg$ A	Negate(A)
A $\subseteq$ B	NegationSearch(A) $\Rightarrow$ NegationSearch(B)
$\exists$ R.C	$\exists$ R.NegationSearch(C)
$\forall$ R.C	$\forall$ R.NegationSearch(C)
A $\equiv$ B	NegationSearch(A) $\equiv$ NegationSearch(B)

Table 4.2: NegationSearch Algorithm

Input	Output
A	$\neg$ A
A $\wedge$ B	Negate(A) $\vee$ Negate(B)
A $\vee$ B	Negate(A) $\wedge$ Negate(B)
$\neg$ A	NegationSearch(A)
A $\subseteq$ B	NegationSearch(A) $\wedge$ Negate(B)
$\exists$ R.C	$\forall$ R.Negate(C)
$\forall$ R.C	$\exists$ R.Negate(C)

Table 4.3: Negate Algorithm

Negation can now only appear as a directly above a concept.

4.3.2 Relocating Universals

Univerals are move so that they only appear as defined to unique concepts.
In all axioms, we make the following changes.

- If $\forall$ R.C is in a statement and C is neither an atomic concept nor the negation of an atomic concept, replace C with X, and add the rule X $\equiv$ C
- If $\forall$ R.C is in a statement and is not of the form $\alpha \equiv \forall$ R.C, we generate a new concept Y, replace $\alpha \equiv \forall$ R.C with $\alpha \equiv$ Y, and add the rule Y $\equiv \forall$ R.C

4.3.3 Translation into GF$^{\neg}$

We add variables and simply defined as statements. This is now in the GF$^{\neg}$ format.

Our translation from Description Logic to First Order Logic is defined recursively by two functions. The first takes a free variable and a concept in description logic. The second takes two free variables and a role in description logic. The Description Logic input formulas are defined recursively in definition 19.

Input	Output
A, x	A(x)
¬A, x	¬A(x)
A ∧ B, x	T_c(A,x) ∧ T_c(B,x)
A ∨ B, x	T_c(A,x) ∨ T_c(B,x)
A ⊆ B, x	T_c(A,x) ⇒ T_c(B,x)
∃ R.C, x	∃ y. T_r(R,x,y) ∧ T_c(C,y)
A ≡ ∀ R.C, x	A(x) ∧ T_r(x,y) ⇒ C(y) ⊤ ⇒ A(x) ∨ ∃ y.R(x,y) ∧ ¬C(y)
A ≡ B, x	T_c(A,x) ⇒ T_c(B,x) T_c(B,x) ⇒ T_c(A,x)

Table 4.4: T_c Algorithm

Input	Output
R,x,y	R(x,y)

Table 4.5: T_r Algorithm

4.3.4 Hypothesis Disjunction Removal

We remove all disjunctions from the hypothesis of a sentence.

Additionally at this stage, we convert all sentence that are not an implication relationship into an implication realtionship with a hypothesis and a conclusion for technical procesing purposes. The only change from this step is that a statement α without an implication relationship is now replaced with $\top \Rightarrow \alpha$.
If a sentence contains a disjunction in the hypothesis: A ∨ B ⇒ C, we can rewrite it as two new sentences:

- A ⇒ C
- B ⇒ C

4.3.5 Negated Concept Removal

We remove all remaining negated concepts. This translates the sentence from $GF^{\neg}$ into GF. There are two different parts, one that deals with negated concepts in the hypothesis and the other in the conclusion.

1. If a negated concept $\neg A$ occurs in the hypothesis of a sentence, then it is at the top level, as in : $\alpha \wedge \neg A \Rightarrow \beta$. Transform this to $\alpha \Rightarrow \beta \vee A$
2. If a negated atom ¬ A occurs in the conclusion of a sentence, then replace it with some unused concept name NegA. Then add the axiom A ∧ NegA ⇒ ⊥ for each relation name A.

4.3.6 Proof of Correctness

Proof of correctness for a translation requires:

1. The algorithm always terminates.
2. All models are preserved by the translation algorithm.

Proof. Proof of termination of the algorithm is very simple, since our algorithm is a constant number of applications of depth first search upon multiple abstract syntax trees. Each step in is a depth algorithm that can only delve further into the abstract syntax tree, reach a leaf node, or add a new tree to the list of trees.

- If any algorithm reaches a leaf node, specifically a role or a concept, then it will clearly terminate.
- If the any tree delving algorithm delves further into the abstract syntax tree, it will clearly terminate if deeper steps terminate.
- If the algorithm adds a new tree to the list of trees, it will clearly terminate since there is now nothing except a new concept to replace the removed tree. Since a new tree can only be created from removed segments of a tree, this process itself will not result in infinite additions of trees.

Each single step will be terminate as long as input is a finite abstract syntax tree. Since the algorithm will terminate in all cases, the algorithm always terminates.

To show that models are preserved, we will demonstrate that they are preserved at each step of the algorithm.

- In 4.3.1, the passing of negations downward until they become atomic is standard practice, and it is well known to preserve models.
- In 4.3.2, any instance of replacing an axiom with a concept and then defining that concept as the replaced axiom will preserve models because the sentences before and after are equivalent.
- In 4.3.3, the translation is simply based on the standard definition of Description Logic as defined from First Order Logic except for the $A \equiv \forall R.C$ rule. However, the $A \equiv \forall R.C$ rule can be derived from the standard definition, just with a few manipulations.

  First we break $A \equiv \forall R.C$ into the two separate statements: $A \Rightarrow \forall R.C$ and $\forall R.C \Rightarrow A$

  $A \Rightarrow \forall R.C$ translates from DL into $\forall x,\ A(x) \Rightarrow (\forall y,\ R(x, y) \Rightarrow C(y))$ definitionally.

We can manipulate ∀x A(x) ⇒ (∀y, R(x, y) ⇒ C(y)) through standard manipulations into ∀x, ∀y, (A(x) ∧ R(x, y)) ⇒ C(y)

∀x, ∀y, A(x) ∧ R(x, y)) ⇒ C(y) can be written as ∀x, ∀y, (A(x) ∧ R(x, y) ⇒ C(y))

Which can be writtenas A(x) ∧ R(x, y) ⇒ C(y), our first rule resulting from A ≡ ∀ R.C

∀ R.C ⇒ A translates from DL into ∀x, (∀y, R(x,y) ⇒ C(y)) ⇒ A(x) definitionally.

We can manipulate ∀x, (∀y, R(x,y) ⇒ C(y)) ⇒ A(x) into ∀x, ¬(∀y, R(x,y) ⇒ C(y)) ∨ A(x)

Applying the negation results in ∀x,(∃y, R(x,y) ∧¬C(y)) ∨ A(x)

Writing the implicit hypothesis result in ∀x, ⊤ ⇒ (∃y, R(x,y) ∧¬C(y)) ∨ A(x)

Which can be written as ⊤ ⇒ (∃y, R(x,y) ∧¬C(y)) ∨ A(x), our second rule resulting from A ≡ ∀ R.C

Thus, all translations are derived from the standard definitions of Description Logic and all models are preserved due to the definition property at this step.

- In 4.3.4, there is no actual change, since ⊤ ⇒ α is implicit in any sentence α without an explicit implication relationship. Additionally, splitting a disjunction on the left side of an implication can be shown to preserve models.

  First, we have the statement, A ∨ B ⇒ C.

  It follows that if A is true, then C is true. Similarly, if B is true, then C is true.

  Using this fact, we can simply split A ∨ B ⇒ C into A ⇒ C and B ⇒ C.

  Thus, that such rule can be applied knowing the the models will be preserved.

- In 4.3.5, the first option, dealing with negations in the hypothesis of a sentence, is a simple manipulation, using the fact that $\phi \Rightarrow \psi$ is equivalent to $\neg\phi \vee \psi$ ([18], page 17).

  If $\alpha \wedge \neg A \Rightarrow$ B, then ¬(α ∧ ¬A) ∨ B, by the equivalence above.

  If ¬(α ∧ ¬A) ∨ B, then (¬ α ∨ ¬¬A) ∨ B

  If (¬ α ∨ ¬¬A) ∨ B, then ¬ α ∨ (A ∨ B)

  If ¬ α ∨ (A ∨ B), then α ⇒ (A ∨ B) by reapplying the equivalence above.

  Thus, that such rule can be applied knowing the the models will be preserved.

  The second option, dealing with negations in the conclusion of a sentence, is similar to the changes in step 2. Since the sentence now contains a concept representing the negation of the removed concept and there is now a new sentence that states the removed concept and the negation of the removed concept is impossible, models are preserved.

Since each step preserves models, the algorithm as a whole preserves models.

Since the algorithm always terminates and models are preserved, the algorithm is correct. ///

Chapter 5

A Case Study

5.1 Introduction to Role-Based Access Control

5.1.1 Background

As mentioned in 3.1.1, Description Logic has been applied as a formalization language for Role-Based Access Control (RBAC) [1] [5]. RBAC is a policy specification that allows policy designers to specify user roles, permissions, and objects, and then map user roles to permission sets for acting upon objects. In this way, RBAC offers a level of abstraction away from assigning permissions to specific users. Instead, when a new user is created in the system, they are assigned a user role, which has certain permissions attached to it. RBAC policies are frequently used in company security policies (for limiting who can act upon certain documents) and in eCommerce settings.

5.1.2 Formalizing Role-Based Access Control with Description Logic

Several [1] [5] authors have written about the benefits of formalizing RBAC with Description Logic, as it allows policy designers to add additional rules that normally cannot be specified with permission matrices. A permission matrix or set of user permissions is normally limited to statements about what types of users can act upon certain objects, generally listed as "permit" and "deny" statements. For example, permission matrices do not typically have a way of specifying that a single user should not hold multiple roles, or that files types are disjoint. In contrast, all RBAC policies can be formalized in Description Logic, where permissions are translated to Description Logic roles, and users/objects are translated to Description Logic concepts. Along with traditional "permit" and "deny" statements, new rules can be added to security policies. Additionally, different flavors of Description Logic can provide varying levels of expressability.

In their paper, *Representation and Reasoning on RBAC: A Description Logic Approach*, Zhao *et al.* further emphasize the benefits of a Description Logic formalization of RBAC [1]. First and foremost, they say that formalization allows for a proof of correctness for any given RBAC policy. Their choice of the $\mathcal{ALCQ}$ Description Logic over first order logic as a formalization language is specifically due to Description Logic's decidability and range of computational complexities determined by fragment choice (For a definition of $\mathcal{ALCQ}$, see Definition 19.). These authors present a different method of formalizing RBAC with Description Logic than Dau and Knecthel do in [6], in that Zhao *et al.* are also interested in formalizing a temporal concept of RBAC in the form of "sessions" that a user logs into. Additionally, the authors recognize the temporal component of Description Logic formalizations as a way to enforce dynamic separation of duties, where no more than a certain number of users can be activated in the same session at once. This dynamic separation of duties is an extension of static separation of duties that is enforced whenever different user types are declared to be mutually exclusive in the Description Logic formalization. Finally, the authors present a case study of using the Description Logic reasoning tool RACER [8] to analyze the consistency and outcomes of a sample RBAC policy.

In [5], Junghwa Chae formalizes Role-Based Access Control (RBAC) policies using the $\mathcal{ALCQI}$ fragment of description logic (for a definition of $\mathcal{ALCQI}$, see 19). RBAC policies are frequently seen in both eCommerce applications and for medical databases. Since Description Logic can be used to formalize large taxonomies and hierarchies, it is a nice fit for the task of modelling access control policies. Typically, RBAC policies involving assigning users to roles, and then granting privileges to each type of role. For example, in a medical database, a patient may be assigned the role Patient, and the hospital can specify that Patients may have read access on office visit summaries, but do not have write access. Chae demonstrates that roles, users, and the objects that are acted upon by users can be formalizes in Description Logic concepts. Additionally, actions that may be performed on users, RBAC roles, and objects (such as assign, canRead, activateSession) can be formalized into Description Logic roles. Chae's paper indicates that Description Logic could potentially grow in popularity as the need for well-defined RBAC policies increases alongside eCommerce and online medical database development.

In their paper, *Access Policy Design Supported by FCA Methods*, Frithjof Dau and Martin Knechtel provide a strong argument for the translation of RBAC into Description Logic fragments [6]. They give a case study using a permission matrix (See Figure 5.1.) that is used later in this chapter for our case study. Description Logic provides more flexibility in terms of describing role hierarchies, temporal knowledge, and additional constraints, Dau and Knechtel argue, giving policy designers greater expressive power. We use an approach similar to theirs, whereby we explore different access perspectives ("default deny" vs. "default allow") and use Description Logics to find unintended consequences of RBAC policies.

5.1.3 Relationship to Open- versus Closed-World Assumption

Some policy languages, such as XACML, allow designers to specify if the policy uses the "default deny" or "default allow" perspective. These perspectives determine how to handle a situation in which there is no explicit rule in the permissions about how a user can interact with an object. The default settings often become important when several security policies are integrated into a single application, such as in a university or hospital setting. If a user attempts to access an object that the security policy does not have an explicit rule for, the default specifies whether or not the user will be able to proceed.

The "default deny" and "default allow" perspectives are alalogous to the open- and closed-world assumptions typically discussed in database and ontology design. Such assumptions need to be made while designing a system, because the assumption determines how certain queries will be answered; if a query asks whether a specified individual exists, but the system has no explicit record of them, the system has two ways to respond. In the closed-world assumption, failure to find an explicit record of such a person implies that no such person exists (because every person in a closed-world system is explicitly listed). Therefore, the closed-world system is analogous to the "default deny" perspective of a security policy, where the failure to find an explicit record or permission listing leads to the report of failure or denial of permission. On the other hand, the open-world perspective would report "not enough information" when asked about a person that is not explicitly listed in a database. Similarly, the "default allow" perspective will grant a user permission to access an object if it cannot find an explicit rule preventing a user from accessing it.

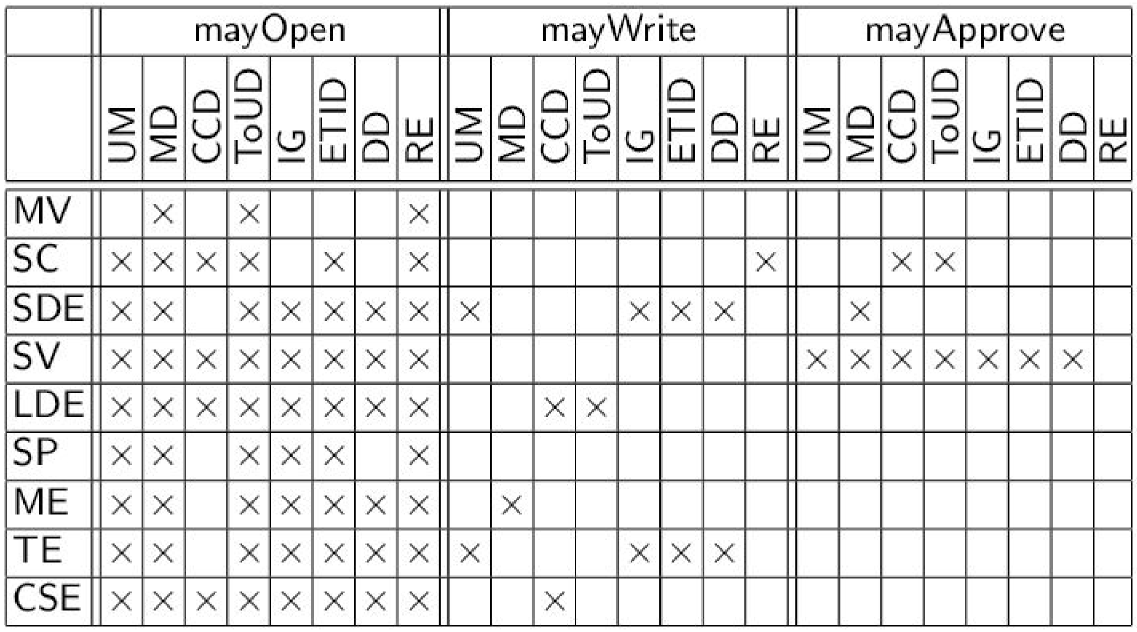
5.2 A Sample Permission Matrix

To demonstrate the benefits of formalizing RBAC in Description Logic, the permission matrix specified in [6] has been translated in full into Description Logic. The permission matrix is shown in Figure 5.1.

The left-hand column lists all possible user types, which are: Marketplace Visitor (MV); Service Consumer (SC); Software Development Engineer (SDE); Service Vendor (SV); Legal Department Employee (LDE); Service Provider (SP); Marketing Employee (ME); Technical Editor (TE); and Customer Service Employee (CSE).

The topmost row lists the three permissions: mayOpen, mayWrite, and mayApprove.

The row below the permissions lists the eight types of documents: User Manual (UM); Marketing Document (MD); Customer Contract Document (CCD); Term of Use Document (ToUD); Installation Guide (IG); External Technical Interface Document (ETID); Design Document (DD); and Rating Entry (RE).



	mayOpen								mayWrite								mayApprove							
	UM	MD	CCD	ToUD	IG	ETID	DD	RE	UM	MD	CCD	ToUD	IG	ETID	DD	RE	UM	MD	CCD	ToUD	IG	ETID	DD	RE
MV		×		×				×																
SC	×	×	×	×		×		×								×			×	×				
SDE	×	×		×	×	×	×	×	×				×	×	×			×						
SV	×	×	×	×	×	×	×	×									×	×	×	×	×	×	×	
LDE	×	×	×	×	×	×	×	×			×	×												
SP	×	×		×	×	×		×																
ME	×	×		×	×	×	×	×		×														
TE	×	×		×	×	×	×	×	×				×	×	×									
CSE	×	×	×	×	×	×	×	×			×													

Figure 5.1: The permission matrix from [6]

5.2.1 Description Logic Formalization

The permission matrix from Figure 5.1 is formalized into Description Logic, as recommended in [6]. This formalization was done by hand, and assumes a "default deny" perspective; that is, the files enumerated for each user type are the files that they are permitted to act upon. The code in 5.1 provides a snapshot of the Description Logic formalization. The full formalization can be found in Appendix section A.1.

Code 5.1: Description Logic Formalization

```
//mayWrite, mayOpen, mayApprove are roles

//***MarketplaceVisitor***
//mayOpen MarketingDocument, TermsDocument, RatingEntry;
//mayWrite nothing;
//mayApprove nothing;cannot
MartketplaceVisitor => forall mayOpen.(MarketingDocument | TermsDocument |
     RatingEntry)
MarketplaceVisitor => forall mayWrite.falsehood & forall mayApprove.
    falsehood
```

5.2.2 Geometric Form Formalization

In addition to formalizing the permission matrix in Figure 5.1 in Description Logic, the permission matrix was also formalized using Geometric Form. This formalization was done so that models could be found using Razor, which only accepts Geometric Form as input.

The first example assumes a “default deny” perspective, and is a hand-translated snapshot of the Description Logic example. The second example assumes a “default allow” perspective, and is also a snapshot that was translated by hand. These first two examples are limited to only examining the MarketplaceVisitor, ServiceVendor, and CustomerService user types. The third example was translated automatically by the algorithm described in Chapter 3. Although all user types were translated automatically, the third example is an excerpt of only the aforementioned three user types to facilitate comparisons. All three examples are reproduced in full in the Appendix in Sections B.1, B.2, and B.3, respectively.

Default Deny Perspective

Code 5.2: Geometric Form Formalization (Default Deny)

```
--------------------------------------------------
-- Type Definition:
--------------------------------------------------
{-There are 3 roles: mayOpen, mayWrite, mayApprove -}
mayOpen(u, f) => User(u) & File(f);
mayWrite(u, f) => User(u) & File(f);
mayApprove(u, f) => User(u) & File(f);
docType(f) = ft => File(f) & Filetype(ft);
userType(u) = ut => User(u) & userType(ut);
--------------------------------------------------
-- Facts:
--------------------------------------------------
{- There are 3 types of users: MarketplaceVisitor, ServiceVendor,
   CustomerService
-}
userType('MarketplaceVisitor);
userType('ServiceVendor);
userType('CustomerService);
{- There are 2 types of documents
-}
docType('UserManual);
docType('MarketingDocument);

{- No other users besides the 3 types are allowed
-}
```

```
userType(ut) => ut = 'MarketplaceVisitor | ut = 'ServiceVendor | ut = '
    CustomerService;

{- No other Filetypes besides the 8 types are allowed
-}
docType(ft) => ft = 'UserManual | ft = 'MarketingDocument;

{- NEW SET OF AXIOMS: docTypes are disjoint -}
'UserManual = 'MarketingDocument => Falsehood;


{- Set the userTypes as disjoint -}
'MarketplaceVisitor = 'ServiceVendor => Falsehood;
'MarketplaceVisitor = 'CustomerService => Falsehood;
'ServiceVendor = 'CustomerService => Falsehood;
--------------------------------------------------
-- Some Axioms:
--------------------------------------------------
{- MarketplaceVisitors can only open MarketingDocument, TermsDocument, and
     RatingEntry
-}
userType(u) = 'MarketplaceVisitor & mayOpen(u, f) => docType(f) = '
    MarketingDocument | docType(f) = 'TermsDocument | docType(f) = '
    RatingEntry;

{- MarketplaceVisitors may not write or approve any Filetypes -}
userType(u) = 'MarketplaceVisitor & mayWrite(u, f) => Falsehood;

userType(u) = 'MarketplaceVisitor & mayApprove(u, f) => Falsehood;
```

Default Allow Perspective

Code 5.3: Geometric Form Formalization (Default Allow)

```
--------------------------------------------------
-- Type Definition:
--------------------------------------------------
{-There are 3 roles: mayOpen, mayWrite, mayApprove -}
mayOpen(u, f) => User(u) & File(f);
mayWrite(u, f) => User(u) & File(f);
mayApprove(u, f) => User(u) & File(f);
docType(f) = ft => File(f) & Filetype(ft);
userType(u) = ut => User(u) & userType(ut);
--------------------------------------------------
-- Facts:
--------------------------------------------------
```

```
{- There are 3 types of users: MarketplaceVisitor, ServiceVendor,
   CustomerService
-}
userType('MarketplaceVisitor);
userType('ServiceVendor);
userType('CustomerService);
{- There are 8 types of documents
-}
docType('UserManual);
docType('MarketingDocument);

{- No other users besides the 3 types are allowed
-}
userType(ut) => ut = 'MarketplaceVisitor | ut = 'ServiceVendor | ut = '
   CustomerService;

{- No other Filetypes besides the 2 types are allowed
-}
docType(ft) => ft = 'UserManual | ft = 'MarketingDocument;

{- Set the userTypes as disjoint -}
'MarketplaceVisitor = 'ServiceVendor => Falsehood;
'MarketplaceVisitor = 'CustomerService => Falsehood;
'ServiceVendor = 'CustomerService => Falsehood;
--------------------------------------------------
-- Some Axioms:
--------------------------------------------------
{-MarketplaceVisitors cannot open UserManual, CustomerContract,
   InstallGuide,
TechInterface, DesignDoc -}

userType(u) = 'MarketplaceVisitor & mayOpen(u,f) & docType(f) = '
   UserManual => Falsehood;
userType(u) = 'MarketplaceVisitor & mayOpen(u,f) & docType(f) = '
   CustomerContract => Falsehood;
```

Automated Translation

Note that the Automated Translation only contains the user type definitions and permission sets for MarketplaceVisitors, ServiceVendors, and CustomerService. No queries are included in this translation. For the full automatic translation, see Appendix Section B.3.

Code 5.4: Geometric Form Formalization (Automated Translation)

```
{-***MarketplaceVisitor***-}
{-mayOpen MarketingDocument, TermsDocument, RatingEntry; -}
{-mayWrite nothing;-}
```

```
{-mayApprove nothing;cannot-}
{-C_1 <=> (ForAll (Role mayOpen) (Or (Concept MarketingDocument) (Or (
    Concept TermsDocument) (Concept RatingEntry))))-}
MartketplaceVisitor(x) => C_1(x);
{-C_2 <=> (ForAll (Role mayOpen) (Or (Concept MarketingDocument) (Or (
    Concept TermsDocument) (Concept RatingEntry))))-}
C_1(x) & mayOpen(x, y2) => C_2(y2);
Truth => C_1(x) | exists y2.mayOpen(x, y2) & !C_2(y2);
C_2(x) => MarketingDocument(x) | TermsDocument(x) | RatingEntry(x);
MarketingDocument(x) => C_2(x);
TermsDocument(x) => C_2(x);
RatingEntry(x) => C_2(x);
{-C_4 <=> (ForAll (Role mayApprove) (Concept falsehood))-}
{-C_3 <=> (ForAll (Role mayWrite) (Concept falsehood))-}
C_4(x) & mayApprove(x, y4) => falsehood(y4);
Truth => C_4(x) | exists y4.mayApprove(x, y4) & !falsehood(y4);
MarketplaceVisitor(x) => C_3(x) & C_4(x);
C_3(x) & mayWrite(x, y6) => falsehood(y6);
Truth => C_3(x) | exists y6.mayWrite(x, y6) & !falsehood(y6);
```

5.3 Modeling in Razor

After translating the permission matrix in Figure 5.1 into Description Logic and finally into Geometric Form, Razor [15] can be employed to run queries over the axioms and to generate models. This section examines various queries run over the Default Deny (5.2) and Default Allow (5.3) Geometric Form translations. Additional queries and their Razor models are provided in Appendix section D.

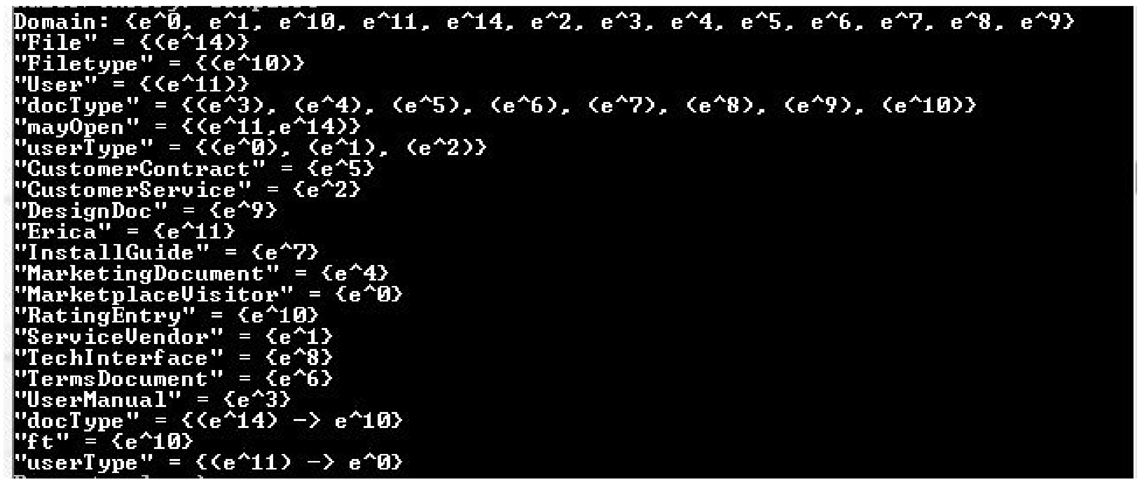
5.3.1 Examining File Types

After designing a security policy, a system engineer may want to test what type of files a specific user type can access. In Razor, testing which files can be opened by a user type is done by writing a query with that user type and file and seeing which types that file can take on. Razor will generate a finite number of minimal models, and those models can be examined to check for any unexpected results. Below is a query that examines what file types a Marketplace Visitor can open:

Code 5.5: File Type Query

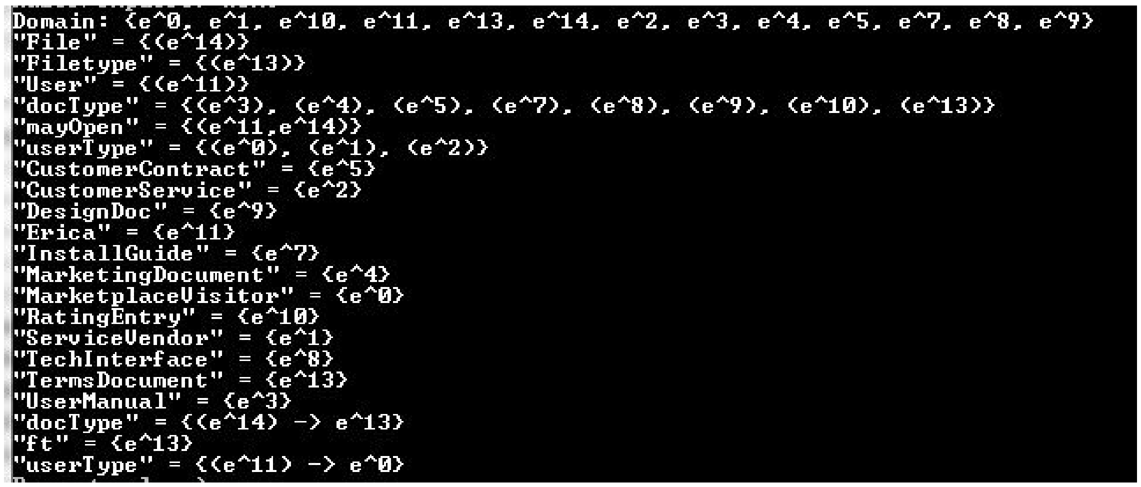
```
{- QUERY: If Erica is a MarketplaceVisitor, what sort of Filetype can
she open? -}
User('Erica);
userType('Erica) = 'MarketplaceVisitor;
docType(‘ft);
exists f . mayOpen('Erica, f) & docType(f) = ‘ft;
```

Razor produces three minimal models, which are presented below:



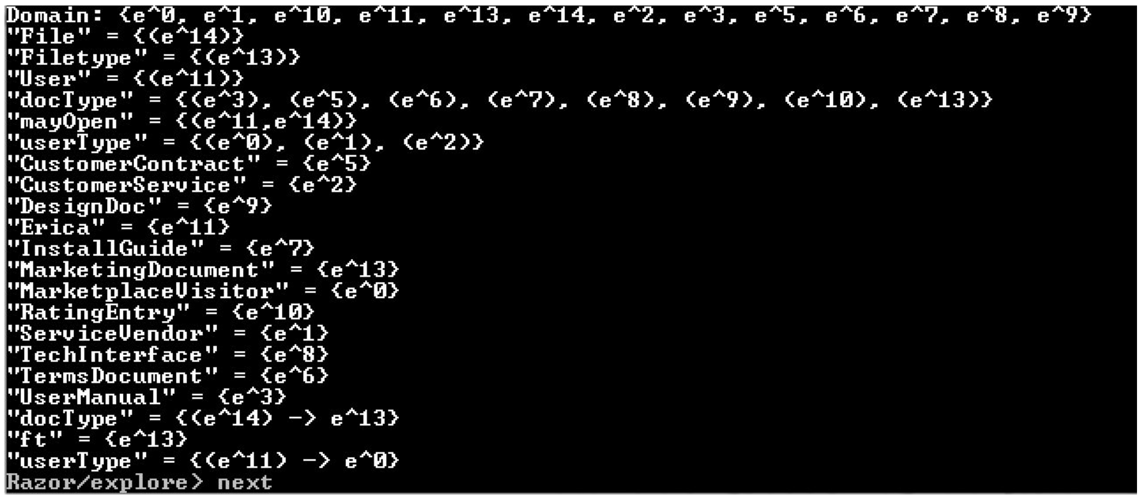
```
Domain: {e^0, e^1, e^10, e^11, e^14, e^2, e^3, e^4, e^5, e^6, e^7, e^8, e^9}
"File" = {(e^14)}
"Filetype" = {(e^10)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^5), (e^6), (e^7), (e^8), (e^9), (e^10)}
"mayOpen" = {(e^11,e^14)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^14) -> e^10}
"ft" = {e^10}
"userType" = {(e^11) -> e^0}
```

Figure 5.2: Erica can open a Rating Entry



```
Domain: {e^0, e^1, e^10, e^11, e^13, e^14, e^2, e^3, e^4, e^5, e^7, e^8, e^9}
"File" = {(e^14)}
"Filetype" = {(e^13)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^5), (e^7), (e^8), (e^9), (e^10), (e^13)}
"mayOpen" = {(e^11,e^14)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^13}
"UserManual" = {e^3}
"docType" = {(e^14) -> e^13}
"ft" = {e^13}
"userType" = {(e^11) -> e^0}
```

Figure 5.3: Erica can open a Terms Document



```
Domain: {e^0, e^1, e^10, e^11, e^13, e^14, e^2, e^3, e^5, e^6, e^7, e^8, e^9}
"File" = {(e^14)}
"Filetype" = {(e^13)}
"User" = {(e^11)}
"docType" = {(e^3), (e^5), (e^6), (e^7), (e^8), (e^9), (e^10), (e^13)}
"mayOpen" = {(e^11,e^14)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^13}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^14) -> e^13}
"ft" = {e^13}
"userType" = {(e^11) -> e^0}
Razor/explore> next
```

Figure 5.4: Erica can open a Marketing Document

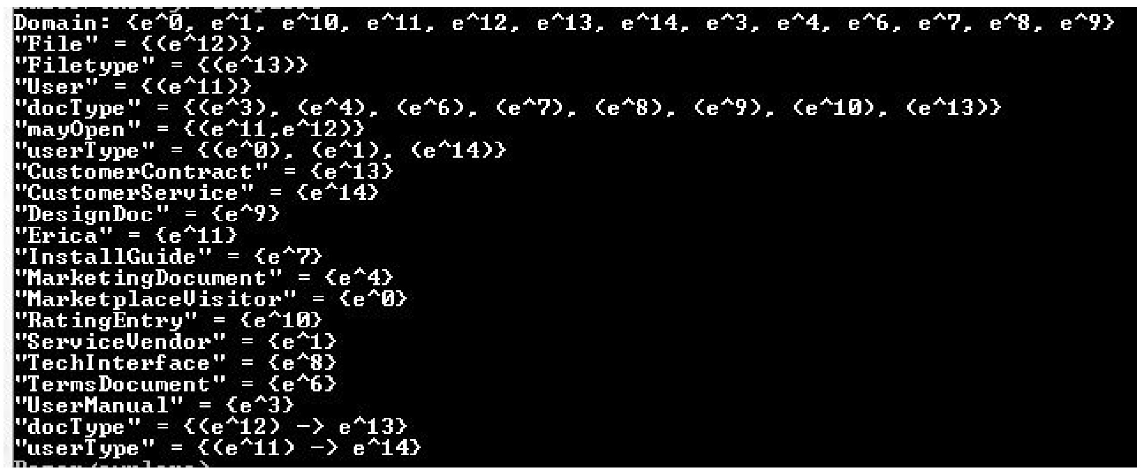
The same results are obtained from using both the “default deny” and “default allow” perspectives.

5.3.2 Examining User Types

A policy designer may also want to check what user types can access a specific file type. For example, if a certain type of document contained confidential information, the policy designer may want to restrict what types of users had permission to access it. Below is a query where a file type is specified, and Razor is queried to see what values a user can take on if the user has permission to open the document:

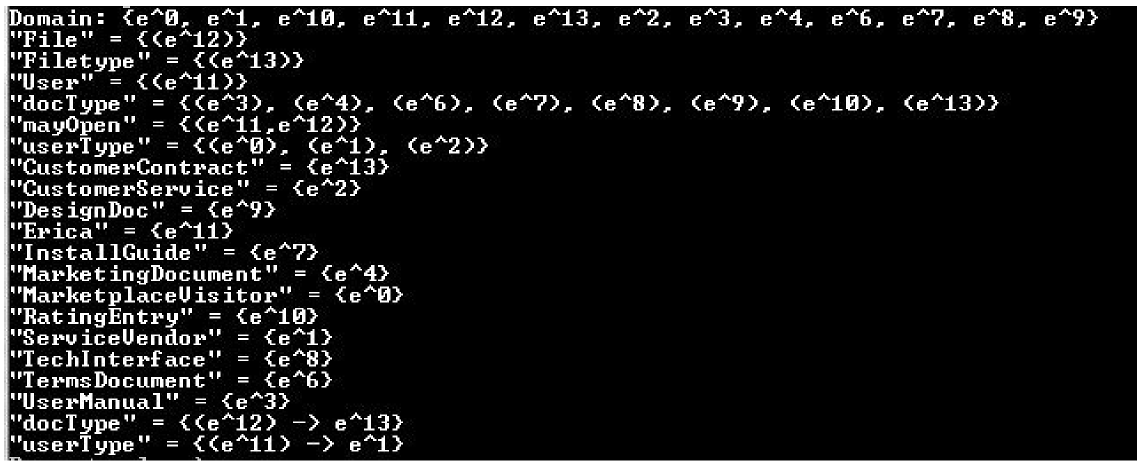
Code 5.6: User Type Query

```
{-Query: If Erica can open a CustomerContract, what type of user is she?
Note: 2 types of users can open a CustomerContract.-}
User('Erica);
exists f . mayOpen('Erica, f) & docType(f) = 'CustomerContract;
exists ut . userType('Erica) = ut;
```



```
Domain: {e^0, e^1, e^10, e^11, e^12, e^13, e^14, e^3, e^4, e^6, e^7, e^8, e^9}
"File" = {(e^12)}
"Filetype" = {(e^13)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^6), (e^7), (e^8), (e^9), (e^10), (e^13)}
"mayOpen" = {(e^11,e^12)}
"userType" = {(e^0), (e^1), (e^14)}
"CustomerContract" = {e^13}
"CustomerService" = {e^14}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^12) -> e^13}
"userType" = {(e^11) -> e^14}
```

Figure 5.5: Erica can be a Customer Service Employee



```
Domain: {e^0, e^1, e^10, e^11, e^12, e^13, e^2, e^3, e^4, e^6, e^7, e^8, e^9}
"File" = {(e^12)}
"Filetype" = {(e^13)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^6), (e^7), (e^8), (e^9), (e^10), (e^13)}
"mayOpen" = {(e^11,e^12)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^13}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^12) -> e^13}
"userType" = {(e^11) -> e^1}
```

Figure 5.6: Erica can be a Service Vendor

Once again, the same results are obtained using the "default deny" and "default allow" perspective.

5.3.3 Multiple User Types Assigned to One User

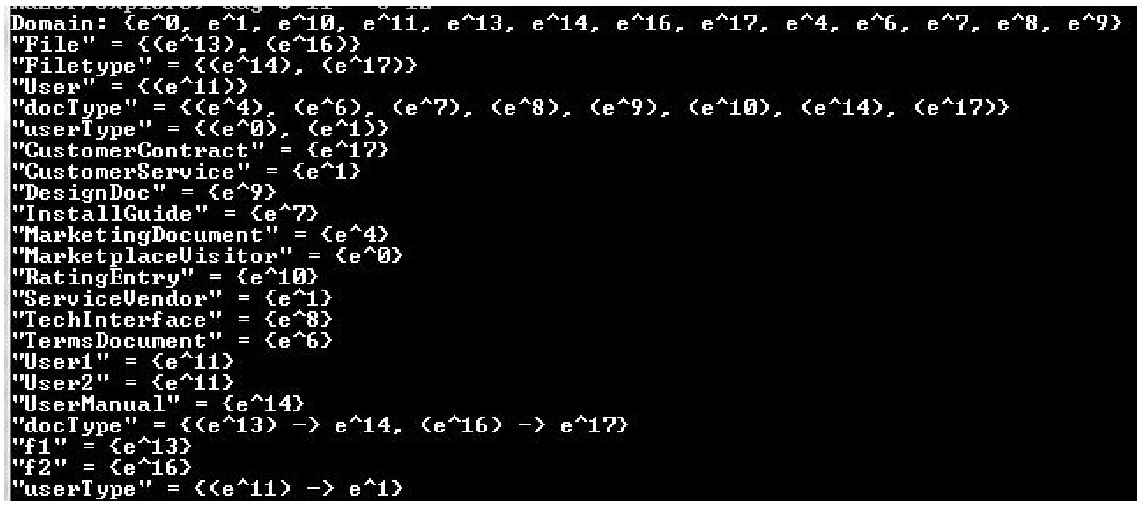
More interesting scenarios arise when dealing with users who are assigned multiple user types. This scenario causes a user to be assigned two separate, sometimes conflicting, permission sets. The code in 5.7 shows the query that is run in both the "default deny" and "default allow" perspectives.

Code 5.7: Multiple User Types Query

```
{- Query: User1 is a MarketplaceVisitor, User2 is a CustomerService
   employee.
Can they be the same person? What files can they open and write?
Run the following model, then in @explore mode,
augment the model by setting the element representing 'User1
equal to the element representing 'User2. Then, try adding the following
    rules:
mayOpen(element representing 'User1, element representing 'f1) and
mayWrite(element representing 'User1, element representing 'f2).-}
User('User1);
User('User2);
File('f1) & docType('f1) = 'UserManual;
File('f2) & docType('f2) = 'CustomerContract;
userType('User1) = 'ServiceVendor;
userType('User2) = 'CustomerService;
```

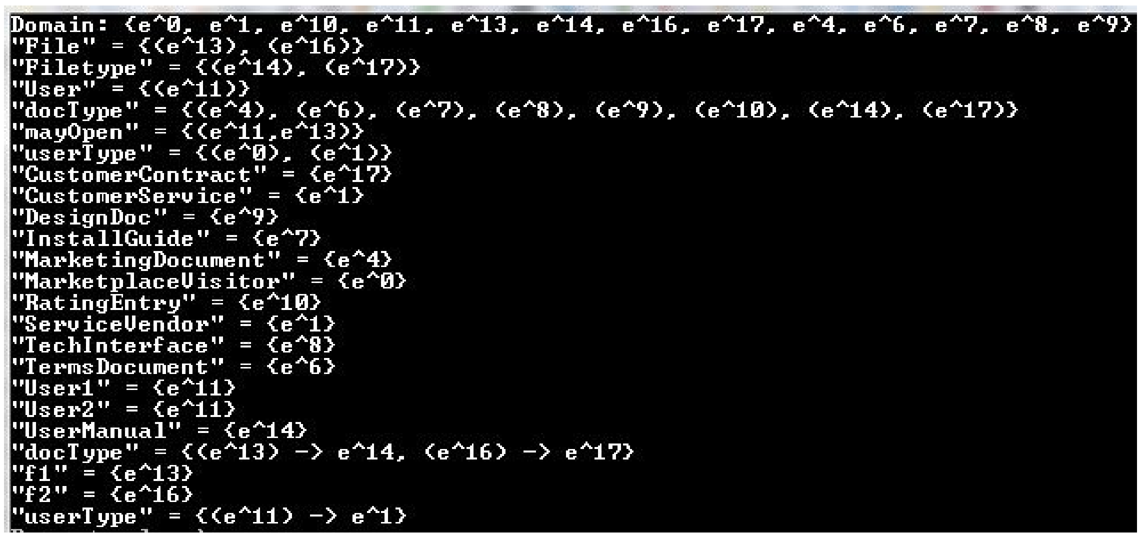
First, let us examine what happens when we run query 5.7 in the "default deny" perspective. We want to know what will happen when a user that has two role types (Customer Service Employee and Service Vendor) attempts to access certain documents. The first document they want to open is a User Manual, which both individual role types have permission to open. The second document they want to write is a Customer Contract, which only Customer Service Employees have permission to write.

In order to run these queries, we first comment out the set of axioms marking the user types as disjoint; once they are not disjoint, we may load the initial set of axioms in Razor. Then, we use Razor's augmentation feature to set the two users equal to each other (giving User1 both user types), which is shown in Figure 5.7. Then, we augment the model by adding the statement that User1 may open a User Manual. The result of this augmentation is shown in Figure 5.8.



```
Domain: {e^0, e^1, e^10, e^11, e^13, e^14, e^16, e^17, e^4, e^6, e^7, e^8, e^9}
"File" = {(e^13), (e^16)}
"Filetype" = {(e^14), (e^17)}
"User" = {(e^11)}
"docType" = {(e^4), (e^6), (e^7), (e^8), (e^9), (e^10), (e^14), (e^17)}
"userType" = {(e^0), (e^1)}
"CustomerContract" = {e^17}
"CustomerService" = {e^1}
"DesignDoc" = {e^9}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"User1" = {e^11}
"User2" = {e^11}
"UserManual" = {e^14}
"docType" = {(e^13) -> e^14, (e^16) -> e^17}
"f1" = {e^13}
"f2" = {e^16}
"userType" = {(e^11) -> e^1}
```

Figure 5.7: Give User1 2 Roles: Service Vendor and Customer Service Employee



```
Domain: {e^0, e^1, e^10, e^11, e^13, e^14, e^16, e^17, e^4, e^6, e^7, e^8, e^9}
"File" = {(e^13), (e^16)}
"Filetype" = {(e^14), (e^17)}
"User" = {(e^11)}
"docType" = {(e^4), (e^6), (e^7), (e^8), (e^9), (e^10), (e^14), (e^17)}
"mayOpen" = {(e^11,e^13)}
"userType" = {(e^0), (e^1)}
"CustomerContract" = {e^17}
"CustomerService" = {e^1}
"DesignDoc" = {e^9}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"User1" = {e^11}
"User2" = {e^11}
"UserManual" = {e^14}
"docType" = {(e^13) -> e^14, (e^16) -> e^17}
"f1" = {e^13}
"f2" = {e^16}
"userType" = {(e^11) -> e^1}
```

Figure 5.8: User1 is allowed to open a User Manual

Next, we augment the model by adding the statement that User1 may write a Customer Contract. However, we are told that no models exist where User1 may write a Customer Contract. This result stems from the fact that, while Customer Service Employees may write Customer Contracts, Service Vendors may not. In this "default deny" perspective, the permissions conflict leads to User1 being denied the ability to write the document. Effectively, this gives any user with multiple user types only the intersection of the two sets of privileges.

We can also run this same set of axioms using the "default allow" perspective. Intuitively, one may assume that if a user has two user types that grant conflicting permissions on a document, the "default allow" perspective would grant the user permission to act upon the object. However, model finders such as Razor are checking sets of axioms for consistency. When a model finder runs into two conflicting sets of axioms, it assumes that a logical error has been made and reports no possible models. Therefore, even in the "default allow" perspective, users with two permission sets are only granted the intersection of those two sets of privileges. Because model finders normally do not support conflict resolution for inconsistent axioms, any sort of permission conflict resolution would need to be manually coded as Geometric Form axioms.

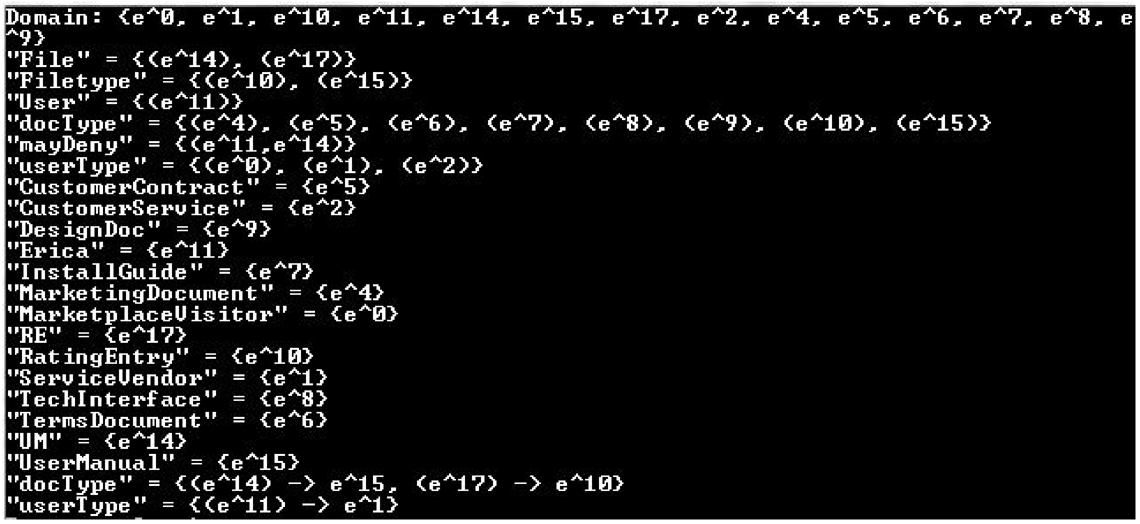
5.3.4 Role Subsumption

One of the advantages of formalizing RBAC using Description Logic is the ability to describe role subsumption, as noted in [6] and [1]. Role subsumption involves the inheritance of permissions by users; for example, being assigned to one user type can grant that user the permission set of that specific user type as well as any permissions held by user types lower in the hierarchy than them.

In the following query, the role subsumption statement is made that any user that may approve a file may also deny that file. Next, Razor is asked to provide a model where the user may deny a file that they may approve. The query is shown in 5.8 and the resulting Razor model is shown in Figure 5.9.

Code 5.8: Role Subsumption Query

```
{- Query: What happens if we allow role subsumption? That is,
having permission to approve a file also gives you permission to
deny that file.
Note: Run the base set of axioms, then see if 'Erica can deny the
   UserManual.-}
mayApprove(u, f) => mayDeny(u,f);
mayDeny(u,f) => User(u) & File(f);
User('Erica) & userType('Erica) = 'ServiceVendor;
File('UM) & docType('UM) = 'UserManual;
mayDeny('Erica, 'UM);
```



```
Domain: {e^0, e^1, e^10, e^11, e^14, e^15, e^17, e^2, e^4, e^5, e^6, e^7, e^8, e
^9}
"File" = {(e^14), (e^17)}
"Filetype" = {(e^10), (e^15)}
"User" = {(e^11)}
"docType" = {(e^4), (e^5), (e^6), (e^7), (e^8), (e^9), (e^10), (e^15)}
"mayDeny" = {(e^11,e^14)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RE" = {e^17}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UM" = {e^14}
"UserManual" = {e^15}
"docType" = {(e^14) -> e^15, (e^17) -> e^10}
"userType" = {(e^11) -> e^1}
```

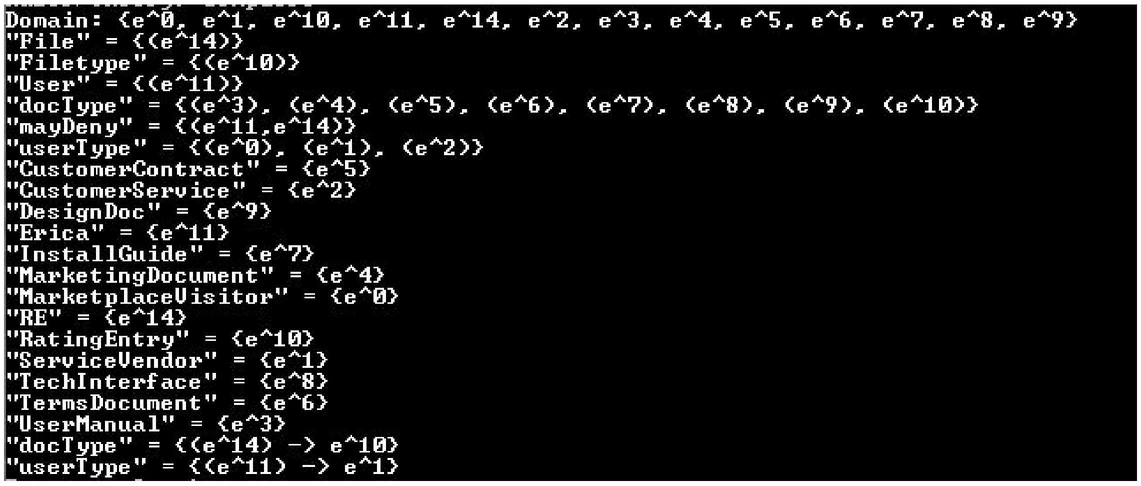
Figure 5.9: Erica is allowed to deny a User Manual

Therefore, we see that, as expected, the user may deny a file (the User Manual) that they already had permission to approve. Note that this result occurs both in the "default deny" and "default allow" perspectives.

In the next query, role subsumption is again examined. In this query, Razor is asked to provide a model where the user may deny a file that they may *not* approve. The query is shown in 5.9 and the resulting Razor model is shown in Figure 5.10.

Code 5.9: Role Subsumption Query

```
{- Query: What happens if we allow role subsumption? That is,
having permission to approve a file also gives you permission to
deny that file.
Note: Run the base set of axioms, then see if 'Erica can deny the
   UserManual.-}
mayApprove(u, f) => mayDeny(u,f);
mayDeny(u,f) => User(u) & File(f);
User('Erica) & userType('Erica) = 'ServiceVendor;
File('RE) & docType('RE) = 'RatingEntry;
mayDeny('Erica, 'RE);
```



```
Domain: {e^0, e^1, e^10, e^11, e^14, e^2, e^3, e^4, e^5, e^6, e^7, e^8, e^9}
"File" = {(e^14)}
"Filetype" = {(e^10)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^5), (e^6), (e^7), (e^8), (e^9), (e^10)}
"mayDeny" = {(e^11,e^14)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RE" = {e^14}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^14) -> e^10}
"userType" = {(e^11) -> e^1}
```

Figure 5.10: Erica is allowed to deny a Rating Entry

Razor points out that, although the user may not approve the Rating Entry, the user may deny it. This result stems from the fact that Razor was only provided with the logical axiom that a user who may approve a document that may deny it. That axiom does not imply the inverse, i.e.: that a user that may not approve a document may not deny that document. Note that this result occurs both in the "default deny" and "default allow" perspectives.

Chapter 6

Conclusion

Description logics form a family of knowledge representation languages for modeling ontologies. Model-finding is a technique for analyzing a first-order theory T by constructing and querying the models of T. This project develops a translation from description logic theories to first-order theories, enabling the use of the Razor model-finder. We provide an implementation of the translation algorithm, and a proof of correctness of the algorithm. As a case study, we explore a sample role-based access control policy formalized in description logic and show how to reason about using Razor. Some future work for after our project could be to implement translation of richer fragments of DL than ALC with atomic roles. More case studies could be performed on RBAC ontologies and more case studies could be performed in ontologies from the TONES repository. Finally, our translation algorithm could be integrated into the Razor application, allowing users to input in the OWL format, Description Logic, and the current Razor input format.

Bibliography

[1] Representation and reasoning on RBAC: A Description Logic Approach. In Dang Van Hung and Martin Wirsing, editors, *Theoretical Aspects of Computing ICTAC 2005*, volume 3722 of *Lecture Notes in Computer Science*. 2005.

[2] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. *Journal of Computer Security*, 2014. In press.

[3] Johannes Bauer. *Model Exploration to Support Understanding of Ontologies*. PhD thesis, Dresden University of Technology, march 2009.

[4] Johannes Bauer, Ulrike Sattler, and Bijan Parsia. Explaining by example: Model exploration for ontology comprehension. In *Description Logics'09*, pages –1–1, 2009.

[5] Junghwa Chae. Modeling of the Role-Based Access Control Policy with constraints using Description Logic. In Osvaldo Gervasi and Marina L. Gavrilova, editors, *Computational Science and Its Applications - ICCSA 2007*, volume 4705 of *Lecture Notes in Computer Science*, pages 500–511. Springer Berlin Heidelberg, 2007.

[6] Frithjof Dau and Martin Knechtel. Access policy design supported by FCA methods. In Sebastian Rudolph, Frithjof Dau, and SergeiO. Kuznetsov, editors, *Conceptual Structures: Leveraging Semantic Technologies*, volume 5662 of *Lecture Notes in Computer Science*, pages 141–154. Springer Berlin Heidelberg, 2009.

[7] Bernardo Cuenca Grau, Ian Horrocks, Boris Motik, Bijan Parsia, Peter Patel-Schneider, and Ulrike Sattler. OWL 2: The next step for OWL. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6:309–322, 2008.

[8] Volker Haarselv, Ralf Moeller, and Michael Wessel. Racer. http://ifis.uni-luebeck.de/index.php?id=385, May 2014.

[9] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible $\mathcal{SROIQ}$. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proc. of the 10th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2006)*, pages 57–67. AAAI Press, 2006.

[10] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for very expressive Description Logics. *Logic Journal of the IGPL*, 8(3):239–263, 2000.

[11] Daniel Jackson. *Software Abstractions-Logic, Language, and Analysis*. MIT Press, Cambridge, MA, 2012.

[12] Daniel Jackson. Alloy: a language and tool for relational models. http://alloy.mit.edu/alloy/, September 2014.

[13] Carsten Lutz. An improved NExpTime-hardness result for Description Logic $\mathcal{ALC}$ extended with inverse roles, nominals, and counting. LTCS-Report 05-05, Chair for Automata Theory, Institute for Theoretical Computer Science, Dresden University of Technology, Dresden, Germany, 2005.

[14] Bijan Parsia, Sebastian Rudolph, Markus Krötzsch, Peter Patel-Schneider, and Pascal Hitzler. OWL 2 web ontology language primer (second edition). Technical report, W3C, December 2012. http://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[15] Salman Saghafi. Razor website. http://salmans.github.io/Razor/, March 2015.

[16] Ulrike Sattler. Description Logic Reasoners. http://www.cs.man.ac.uk/~sattler/reasoners.html, July 2010.

[17] Manfred Schmidt-Schaußand Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[18] Dirk van Dalen. *Logic and Structure*. Springer-Verlag, London, 5 edition, 2013.

[19] Christopher Welty and Deborah McGuinness. OWL web ontology language guide. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[20] Evgeny Zolin. Complexity of reasoning inDescription Logics. http://www.cs.man.ac.uk/~ezolin/dl/, August 2013.

Appendices

Appendix A

Description Logic Examples

A.1 RBAC Example

```
//RBAC.dl example of a company's permission system
//Modelled after the permission matrix on page 4 of http://citeseerx.ist.
   psu.edu/viewdoc/download;jsessionid=B1B59456CE9702EB37414FDFD98AEAB2?
   doi=10.1.1.157.9388&rep=rep1&type=pdf

//mayWrite, mayOpen, mayApprove are roles

//***MarketplaceVisitor***
//mayOpen MarketingDocument, TermsDocument, RatingEntry;
//mayWrite nothing;
//mayApprove nothing;cannot
MartketplaceVisitor => forall mayOpen.(MarketingDocument | TermsDocument |
    RatingEntry)
MarketplaceVisitor => forall mayWrite.falsehood & forall mayApprove.
   falsehood

//***ServiceConsumer***
//mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
   RatingEntry, TechInterface;
//mayWrite RatingEntry;
//mayApprove CustomerContract, TermsDocument;
ServiceConsumer => forall mayOpen.(UserManual | MarketingDocument |
   CustomerContract | TermsDocument | RatingEntry | TechInterface)
ServiceConsumer => forall mayWrite.(RatingEntry)
ServiceConsumer => forall mayApprove.(CustomerContract | TermsDocument)

//***SoftwareEng***
//mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   RatingEntry, TechInterface, DesignDoc;
```

```
//mayWrite UserManual, TechInterface, InstallGuide, DesignDoc;
//mayApprove MarketingDocument
SoftwareEng => forall mayOpen.(UserManual | MarketingDocument |
   TermsDocument | InstallGuide | RatingEntry | TechInterface | DesignDoc
   )
SoftwareEng => forall mayWrite.(UserManual | TechInterface | InstallGuide
   | DesignDoc)
SoftwareEng => forall mayApprove.(MarketingDocument)

//***ServiceVendor***
//mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
   InstallGuide, TechInterface, DesignDoc, RatingEntry;
//mayWrite nothing;
//mayApprove UserManual, MarketingDocument, CustomerContract,
   TermsDocument, InstallGuide, TechInterface, DesignDoc;
ServiceVendor => forall mayOpen.(UserManual | MarketingDocument |
   CustomerContract | TermsDocument | InstallGuide | TechInterface |
   DesignDoc | RatingEntry)
ServiceVendor => forall mayWrite.falsehood
ServiceVendor => forall mayApprove.(UserManual | MarketingDocument |
   CustomerContract | TermsDocument | InstallGuide | TechInterface |
   DesignDoc)

//***LegalDept***
//mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
   InstallGuide, TechInterface, DesignDoc, RatingEntry;
//mayWrite CustomerContract, TermsDocument;
//mayApprove nothing;
LegalDept => forall mayOpen.(UserManual | MarketingDocument |
   CustomerContract | TermsDocument | InstallGuide | TechInterface |
   DesignDoc | RatingEntry)
LegalDept => forall mayWrite.(CustomerContract | TermsDocument)
LegalDept => forall mayApprove.falsehood

//***ServiceProvider****
//mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   TechInterface, RatingEntry;
//mayWrite nothing;
//mayApprove nothing;
ServiceProvider => forall mayOpen.(UserManual | MarketingDocument |
   TermsDocument | InstallGuide | TechInterface | RatingEntry)
ServiceProvider => forall mayWrite.falsehood & forall mayApprove.falsehood

//***Marketing***
```

```
//mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   TechInterface, DesignDoc, RatingEntry;
//mayWrite MarketingDocument;
//mayApprove nothing;
Marketing => forall mayOpen.(UserManual | MarketingDocument |
   TermsDocument | InstallGuide | TechInterface | DesignDoc | RatingEntry
   )
Marketing => forall mayWrite.MarketingDocument
Marketing => forall mayApprove.falsehood

//***TechEditor***
//mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   TechInterface, DesignDoc, RatingEntry;
//mayWrite UserManual, InstallGuide, TechInterface, DesginDoc;
//mayApprove nothing;
TechEditor => forall mayOpen.(UserManual | MarketingDocument |
   TermsDocument | InstallGuide | TechInterface | DesignDoc | RatingEntry
   )
TechEditor => forall mayWrite.(UserManual | InstallGuide | TechInterface |
    DesginDoc)
TechEditor => forall mayApprove.falsehood

//***CustomerService***
//mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
   InstallGuide, TechInterface, DesignDoc, RatingEntry;
//mayWrite CustomerContract
//mayApprove nothing;
CustomerService => forall mayOpen.(UserManual | MarketingDocument |
   CustomerContract | TermsDocument | InstallGuide | TechInterface |
   DesignDoc | RatingEntry)
CustomerService => forall mayWrite.CustomerContract
CustomerService => forall mayApprove.falsehood
```

Appendix B

Geometric Logic Examples

B.1 Default Deny

```
{- Razor’s Sample Specification
Filetype : defaultdeny.raz
Description: A small formalization in GL of the rbac.dl example.
Written up by hand.
** Please uncomment one query at a time before running Razor on this
specification.
-}
--------------------------------------------------
-- Type Definition:
--------------------------------------------------
{-There are 3 roles: mayOpen, mayWrite, mayApprove -}
mayOpen(u, f) => User(u) & File(f);
mayWrite(u, f) => User(u) & File(f);
mayApprove(u, f) => User(u) & File(f);
docType(f) = ft => File(f) & Filetype(ft);
userType(u) = ut => User(u) & userType(ut);
--------------------------------------------------
-- Facts:
--------------------------------------------------
{- There are 3 types of users: MarketplaceVisitor, ServiceVendor,
   CustomerService
-}
userType('MarketplaceVisitor);
userType('ServiceVendor);
userType('CustomerService);
{- There are 8 types of documents
-}
docType('UserManual);
docType('MarketingDocument);
docType('CustomerContract);
docType('TermsDocument);
docType('InstallGuide);
docType('TechInterface);
docType('DesignDoc);
docType('RatingEntry);

{- No other users besides the 3 types are allowed
-}
userType(ut) => ut = 'MarketplaceVisitor | ut = 'ServiceVendor | ut = '
   CustomerService;

{- No other Filetypes besides the 8 types are allowed
-}
docType(ft) => ft = 'UserManual | ft = 'MarketingDocument | ft = '
   CustomerContract | ft = 'TermsDocument
| ft = 'InstallGuide | ft = 'TechInterface | ft = 'DesignDoc | ft = '
   RatingEntry | ft = 'CustomerData;

{- NEW SET OF AXIOMS: docTypes are disjoint -}
'UserManual = 'MarketingDocument => Falsehood;
'UserManual = 'CustomerContract => Falsehood;
'UserManual = 'TermsDocument => Falsehood;
'UserManual = 'InstallGuide => Falsehood;
'UserManual = 'TechInterface => Falsehood;
'UserManual = 'DesignDoc => Falsehood;
'UserManual = 'RatingEntry => Falsehood;
'MarketingDocument = 'CustomerContract => Falsehood;
'MarketingDocument = 'TermsDocument => Falsehood;
'MarketingDocument = 'InstallGuide => Falsehood;
'MarketingDocument = 'TechInterface => Falsehood;
'MarketingDocument = 'DesignDoc => Falsehood;
'MarketingDocument = 'RatingEntry => Falsehood;
'CustomerContract = 'TermsDocument => Falsehood;
'CustomerContract = 'InstallGuide => Falsehood;
'CustomerContract = 'TechInterface => Falsehood;
'CustomerContract = 'DesignDoc => Falsehood;
'CustomerContract = 'RatingEntry => Falsehood;
'TermsDocument = 'InstallGuide => Falsehood;
'TermsDocument = 'TechInterface => Falsehood;
'TermsDocument = 'DesignDoc => Falsehood;
'TermsDocument = 'RatingEntry => Falsehood;
'InstallGuide = 'TechInterface => Falsehood;
'InstallGuide = 'DesignDoc => Falsehood;
'InstallGuide = 'RatingEntry => Falsehood;
```

```
'TechInterface = 'DesignDoc => Falsehood;
'TechInterface = 'RatingEntry => Falsehood;
'DesignDoc = 'RatingEntry => Falsehood;
'CustomerData = 'UserManual => Falsehood;
'CustomerData = 'MarketingDocument => Falsehood;
'CustomerData = 'CustomerContract => Falsehood;
'CustomerData = 'TermsDocument => Falsehood;
'CustomerData = 'InstallGuide => Falsehood;
'CustomerData = 'TechInterface => Falsehood;
'CustomerData = 'DesignDoc => Falsehood;
'CustomerData = 'RatingEntry => Falsehood;

{- Set the userTypes as disjoint -}
'MarketplaceVisitor = 'ServiceVendor => Falsehood;
'MarketplaceVisitor = 'CustomerService => Falsehood;
'ServiceVendor = 'CustomerService => Falsehood;
--------------------------------------------------
-- Some Axioms:
--------------------------------------------------
{- MarketplaceVisitors can only open MarketingDocument, TermsDocument, and
    RatingEntry
-}
userType(u) = 'MarketplaceVisitor & mayOpen(u, f) => docType(f) = '
   MarketingDocument | docType(f) = 'TermsDocument | docType(f) = '
   RatingEntry;

{- MarketplaceVisitors may not write or approve any Filetypes -}
userType(u) = 'MarketplaceVisitor & mayWrite(u, f) => Falsehood;

userType(u) = 'MarketplaceVisitor & mayApprove(u, f) => Falsehood;

{- ServiceVendors may Open UserManual, MarketingDocument, CustomerContract
   , TermsDocument,
InstallGuide, TechInterface, DesignDoc, RatingEntry -}
userType(u) = 'ServiceVendor & mayOpen(u, f) => docType(f) = 'UserManual |
    docType(f) = 'MarketingDocument | docType(f) = 'CustomerContract |
docType(f) = 'TermsDocument | docType(f) = 'InstallGuide | docType(f) = '
   TechInterface | docType(f) =' DesignDoc | docType(f) = 'RatingEntry;

{- ServiceVendors mayWrite nothing -}
userType(u) = 'ServiceVendor & mayWrite(u, f) => Falsehood;

{- ServiceVendors may Approve UserManual, MarketingDocument,
   CustomerContract,
TermsDocument, InstallGuide, TechInterface, DesignDoc -}
```

```
userType(u) = 'ServiceVendor & mayApprove(u, f) => docType(f) = '
   UserManual | docType(f) = 'MarketingDocument | docType(f) = '
   CustomerContract |
docType(f) = 'TermsDocument | docType(f) = 'InstallGuide | docType(f) = '
   TechInterface | docType(f) = 'DesignDoc;

{- CustomerService may Open UserManual, MarketingDocument,
   CustomerContract, TermsDocument,
InstallGuide, TechInterface, DesignDoc, RatingEntry; -}
userType(u) = 'CustomerService & mayOpen(u, f) => docType(f) = 'UserManual
    | docType(f) = 'MarketingDocument | docType(f) = 'CustomerContract |
docType(f) = 'TermsDocument | docType(f) = 'InstallGuide | docType(f) = '
   TechInterface | docType(f) = 'DesignDoc | docType(f) = 'RatingEntry;

{- CustomerService may Write CustomerContract -}
userType(u) = 'CustomerService & mayWrite(u, f) => docType(f) = '
   CustomerContract;

{- CustomerService may not approve anything -}
userType(u) = 'CustomerService & mayApprove(u, f) => Falsehood;


--------------------------------------------------
-- Queries:
--------------------------------------------------
{- QUERY 1: If Erica is a MarketplaceVisitor, what sort of Filetype can
she open? -}
--User('Erica);
--userType('Erica) = 'MarketplaceVisitor;
--exists f . mayOpen('Erica, f);

--------------------------------------------------
{- Query 2: If Erica is CustomerService, can she approve a
   CustomerContract?
-}
--User('Erica);
--userType('Erica) = 'CustomerService;
--exists c . mayApprove('Erica, c);

--------------------------------------------------
{-Query 3: If Erica writes a CustomerContract, what type of user is she?
Note: Only 1 type of user can write a CustomerContract.-}
--User('Erica);
--exists f . mayWrite('Erica, f) & docType(f) = 'CustomerContract;
--exists ut . userType('Erica) = ut;
```

```
--------------------------------------------------
{-Query 4: If Erica can open a CustomerContract, what type of user is she?
Note: 2 types of users can open a CustomerContract.-}
--User('Erica);
--exists f . mayOpen('Erica, f) & docType(f) = 'CustomerContract;
--exists ut . userType('Erica) = ut;

--------------------------------------------------
{- The following 2 queries test what occurs when a user has 2 permission
   sets. -}
{-Query 5a: If someone writes a CustomerContract and someone approves that
CustomerContract, can they be the same person? Run the following model,
   then, while
in @explore mode, augment the model by setting the element representing '
   User1
equal to the element representing 'User2. -}
--User('User1);
--User('User2);
--File('f) & docType('f) = 'CustomerContract;
--exists ut1 . mayWrite('User1, 'f) & userType('User1) = ut1;
--exists ut2 . mayApprove('User2, 'f) & userType('User2) = ut2;

--------------------------------------------------
{- Query 5b: User1 is a MarketplaceVisitor, User2 is a CustomerService
   employee.
Can they be the same person? What files can they open and write?
Run the following model, then in @explore mode,
augment the model by setting the element representing 'User1
equal to the element representing 'User2. Then, try adding the following
   rules:
mayOpen(element representing 'User1, element representing 'f1) and
mayWrite(element representing 'User1, element representing 'f2).-}
--User('User1);
--User('User2);
--File('f1) & docType('f1) = 'UserManual;
--File('f2) & docType('f2) = 'CustomerContract;
--userType('User1) = 'ServiceVendor;
--userType('User2) = 'CustomerService;

--------------------------------------------------
{- Query 6: What happens if we make user types disjoint? That is,
a user may only have 1 type assigned to it. -}
{- Set the userTypes as disjoint -}
--'MarketplaceVisitor = 'ServiceVendor => Falsehood;
```

```
--'MarketplaceVisitor = 'CustomerService => Falsehood;
--'ServiceVendor = 'CustomerService => Falsehood;
{-Now set up 2 users with 2 separate usertypes. Run this initial
model, then augment the model by setting 'User1 = 'User2. -}
--User('User1);
--User('User2);
--userType('User1) = 'MarketplaceVisitor;
--userType('User2) = 'ServiceVendor;

--------------------------------------------------
{- Query 7: What happens if we allow role subsumption? That is,
having permission to approve a file also gives you permission to
deny that file.
Note: Run the base set of axioms, then see if 'Erica can deny the
   UserManual
and the RatingEntry.-}
--mayApprove(u, f) => mayDeny(u,f);
--mayDeny(u,f) => User(u) & File(f);
--User('Erica) & userType('Erica) = 'ServiceVendor;
--File('UM) & docType('UM) = 'UserManual;
--File('RE) & docType('RE) = 'RatingEntry;
--------------------------------------------------
{- Query 8: What happens if we create a new docType, 'CustomerData?
Can any of the three user types open, write, or approve it?
Note: After adding the new docType, add it to the list of allowable
   docTypes.
Run the initial model, then try augmenting it with mayOpen, mayWrite, and
   mayApprove
for the 3 usertypes and the new docType.-}
--docType('CustomerData);
--User('MV) & userType('MV) = 'MarketplaceVisitor;
--User('SV) & userType('SV) = 'ServiceVendor;
--User('CS) & userType('CS) = 'CustomerService;
--File('f) & docType('f) = 'CustomerData;
```

B.2 Default Allow

```
{- Razor's Sample Specification
Filetype : defaultAllow.raz
Description: A small formalization in GL of the rbac.dl example.
Written up by hand.
** Please uncomment one query at a time before running Razor on this
specification.
-}
--------------------------------------------------
```

```
-- Type Definition:
---------------------------------------------------
{-There are 3 roles: mayOpen, mayWrite, mayApprove -}
mayOpen(u, f) => User(u) & File(f);
mayWrite(u, f) => User(u) & File(f);
mayApprove(u, f) => User(u) & File(f);
docType(f) = ft => File(f) & Filetype(ft);
userType(u) = ut => User(u) & userType(ut);
---------------------------------------------------
-- Facts:
---------------------------------------------------
{- There are 3 types of users: MarketplaceVisitor, ServiceVendor,
   CustomerService
-}
userType('MarketplaceVisitor);
userType('ServiceVendor);
userType('CustomerService);
{- There are 8 types of documents
-}
docType('UserManual);
docType('MarketingDocument);
docType('CustomerContract);
docType('TermsDocument);
docType('InstallGuide);
docType('TechInterface);
docType('DesignDoc);
docType('RatingEntry);

{- No other users besides the 3 types are allowed
-}
userType(ut) => ut = 'MarketplaceVisitor | ut = 'ServiceVendor | ut = '
   CustomerService;

{- No other Filetypes besides the 8 types are allowed
-}
docType(ft) => ft = 'UserManual | ft = 'MarketingDocument | ft = '
   CustomerContract | ft = 'TermsDocument
| ft = 'InstallGuide | ft = 'TechInterface | ft = 'DesignDoc | ft = '
   RatingEntry;

{- NEW SET OF AXIOMS: docTypes are disjoint -}
'UserManual = 'MarketingDocument => Falsehood;
'UserManual = 'CustomerContract => Falsehood;
'UserManual = 'TermsDocument => Falsehood;
'UserManual = 'InstallGuide => Falsehood;
```

```
'UserManual = 'TechInterface => Falsehood;
'UserManual = 'DesignDoc => Falsehood;
'UserManual = 'RatingEntry => Falsehood;
'MarketingDocument = 'CustomerContract => Falsehood;
'MarketingDocument = 'TermsDocument => Falsehood;
'MarketingDocument = 'InstallGuide => Falsehood;
'MarketingDocument = 'TechInterface => Falsehood;
'MarketingDocument = 'DesignDoc => Falsehood;
'MarketingDocument = 'RatingEntry => Falsehood;
'CustomerContract = 'TermsDocument => Falsehood;
'CustomerContract = 'InstallGuide => Falsehood;
'CustomerContract = 'TechInterface => Falsehood;
'CustomerContract = 'DesignDoc => Falsehood;
'CustomerContract = 'RatingEntry => Falsehood;
'TermsDocument = 'InstallGuide => Falsehood;
'TermsDocument = 'TechInterface => Falsehood;
'TermsDocument = 'DesignDoc => Falsehood;
'TermsDocument = 'RatingEntry => Falsehood;
'InstallGuide = 'TechInterface => Falsehood;
'InstallGuide = 'DesignDoc => Falsehood;
'InstallGuide = 'RatingEntry => Falsehood;
'TechInterface = 'DesignDoc => Falsehood;
'TechInterface = 'RatingEntry => Falsehood;
'DesignDoc = 'RatingEntry => Falsehood;

{- Set the userTypes as disjoint -}
--'MarketplaceVisitor = 'ServiceVendor => Falsehood;
-- 'MarketplaceVisitor = 'CustomerService => Falsehood;
-- 'ServiceVendor = 'CustomerService => Falsehood;
--------------------------------------------------
-- Some Axioms:
--------------------------------------------------
{-MarketplaceVisitors cannot open UserManual, CustomerContract,
   InstallGuide,
TechInterface, DesignDoc -}

userType(u) = 'MarketplaceVisitor & mayOpen(u,f) & docType(f) = '
   UserManual => Falsehood;
userType(u) = 'MarketplaceVisitor & mayOpen(u,f) & docType(f) = '
   CustomerContract => Falsehood;
userType(u) = 'MarketplaceVisitor & mayOpen(u,f) & docType(f) = '
   InstallGuide => Falsehood;
userType(u) = 'MarketplaceVisitor & mayOpen(u,f) & docType(f) = '
   TechInterface => Falsehood;
```

```
userType(u) = 'MarketplaceVisitor & mayOpen(u,f) & docType(f) = 'DesignDoc
     => Falsehood;

{- MarketplaceVisitors cannot write anything -}
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   UserManual => Falsehood;
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   MarketingDocument => Falsehood;
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   CustomerContract => Falsehood;
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   TermsDocument => Falsehood;
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   InstallGuide => Falsehood;
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   TechInterface => Falsehood;
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   DesignDoc => Falsehood;
userType(u) = 'MarketplaceVisitor & mayWrite(u,f) & docType(f) = '
   RatingEntry => Falsehood;

{- MarketplaceVisitors cannot approve anything -}
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   UserManual => Falsehood;
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   MarketingDocument => Falsehood;
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   CustomerContract => Falsehood;
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   TermsDocument => Falsehood;
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   InstallGuide => Falsehood;
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   TechInterface => Falsehood;
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   DesignDoc => Falsehood;
userType(u) = 'MarketplaceVisitor & mayApprove(u,f) & docType(f) = '
   RatingEntry => Falsehood;

{-ServiceVendors may open anything. Hence, we don't need any
   specifications to restrict them-}

{-ServiceVendors may not write anything-}
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = 'UserManual =>
     Falsehood;
```

```
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = '
   MarketingDocument => Falsehood;
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = '
   CustomerContract => Falsehood;
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = 'TermsDocument
    => Falsehood;
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = 'InstallGuide
   => Falsehood;
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = 'TechInterface
    => Falsehood;
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = 'DesignDoc =>
   Falsehood;
userType(u) = 'ServiceVendor & mayWrite(u,f) & docType(f) = 'RatingEntry
   => Falsehood;

{-ServiceVendors may not approve RatingEntries -}
userType(u) = 'ServiceVendor & mayApprove(u,f) & docType(f) = 'RatingEntry
    => Falsehood;

{-CustomerService may open anything. Hence, we don't need any
   specifications to restrict them-}

{-CustomerService may not write UserManual, MarketingDocument,
   TermsDocument, InstallGuide,
TechInterface, DesignDoc, RatingEntry -}
userType(u) = 'CustomerService & mayWrite(u,f) & docType(f) = 'UserManual
   => Falsehood;
userType(u) = 'CustomerService & mayWrite(u,f) & docType(f) = '
   MarketingDocument => Falsehood;
userType(u) = 'CustomerService & mayWrite(u,f) & docType(f) = '
   TermsDocument => Falsehood;
userType(u) = 'CustomerService & mayWrite(u,f) & docType(f) = '
   InstallGuide => Falsehood;
userType(u) = 'CustomerService & mayWrite(u,f) & docType(f) = '
   TechInterface => Falsehood;
userType(u) = 'CustomerService & mayWrite(u,f) & docType(f) = 'DesignDoc
   => Falsehood;
userType(u) = 'CustomerService & mayWrite(u,f) & docType(f) = 'RatingEntry
    => Falsehood;

{-CustomerService cannot approve anything -}
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = '
   UserManual => Falsehood;
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = '
   MarketingDocument => Falsehood;
```

```
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = '
   CustomerContract => Falsehood;
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = '
   TermsDocument => Falsehood;
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = '
   InstallGuide => Falsehood;
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = '
   TechInterface => Falsehood;
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = 'DesignDoc
    => Falsehood;
userType(u) = 'CustomerService & mayApprove(u,f) & docType(f) = '
   RatingEntry => Falsehood;

--------------------------------------------------
-- Queries:
--------------------------------------------------
{- QUERY 1: If Erica is a MarketplaceVisitor, what sort of Filetype can
she open? [You get the same results as Query 1 in smallrbac. -}
--User('Erica);
--userType('Erica) = 'MarketplaceVisitor;
--docType('ft);
--exists f. mayOpen('Erica, f) & docType(f) = 'ft;

--------------------------------------------------
{- Query 2: If Erica is CustomerService, can she approve a
   CustomerContract?
[You get the same results as Query 2 in smallrbac.]
-}
--User('Erica);
--userType('Erica) = 'CustomerService;
--docType('ft);
--exists c . mayApprove('Erica, c) & docType(c) = 'ft;

--------------------------------------------------
{-Query 3: If Erica writes a CustomerContract, what type of user is she?
Note: Only 1 type of user can write a CustomerContract. [You get the same
   results
as with Query 3 in smallrbac.-}
--User('Erica);
--exists f . mayWrite('Erica, f) & docType(f) = 'CustomerContract;
--exists ut . userType('Erica) = ut;

--------------------------------------------------
{-Query 4: If Erica can open a CustomerContract, what type of user is she?
```

```
Note: 2 types of users can open a CustomerContract. [You get the same
   results as with
running Query 4 in smallrbac.-}
--User('Erica);
--exists f . mayOpen('Erica, f) & docType(f) = 'CustomerContract;
--exists ut . userType('Erica) = ut;

--------------------------------------------------
{- The following 2 queries test what occurs when a user has 2 permission
   sets. -}

{-Query 5a: If someone writes a CustomerContract and someone approves that
CustomerContract, can they be the same person? Be sure to comment out
the axioms saying the user types are disjoint. Run the following model,
   then, while
in @explore mode, augment the model by setting the element representing '
   User1
equal to the element representing 'User2. -}
--User('User1);
--User('User2);
--File('f) & docType('f) = 'CustomerContract;
--exists ut1 . mayWrite('User1, 'f) & userType('User1) = ut1;
--exists ut2 . mayApprove('User2, 'f) & userType('User2) = ut2;
{-You get the same results by running Query 5a in smallrbac. That is, you
   cannot get
a model when you augment that the 2 users are equal. -}
--------------------------------------------------
{- Query 5b: User1 is a MarketplaceVisitor, User2 is a CustomerService
   employee.
Can they be the same person? What files can they open and write?
Run the following model, then in @explore mode,
augment the model by setting the element representing 'User1
equal to the element representing 'User2. Then, try adding the following
   rules:
mayOpen(element representing 'User1, element representing 'f1) and
mayWrite(element representing 'User1, element representing 'f2).-}
--User('User1);
--User('User2);
--File('f1) & docType('f1) = 'UserManual;
--File('f2) & docType('f2) = 'CustomerContract;
--userType('User1) = 'ServiceVendor;
--userType('User2) = 'CustomerService;
{-The new combined user can open the Usermanual (as seen in smallrbac),
   and also may write
the CustomerContract (which was banned in smallrbac). This shows that if a
     user has 2 user types
they now get the union of the 2 permission sets. -}
--------------------------------------------------
{- Query 6: What happens if we make user types disjoint? That is,
a user may only have 1 type assigned to it. -}
{- Set the userTypes as disjoint -}
--'MarketplaceVisitor = 'ServiceVendor => Falsehood;
--'MarketplaceVisitor = 'CustomerService => Falsehood;
--'ServiceVendor = 'CustomerService => Falsehood;
{-Now set up 2 users with 2 separate usertypes. Run this initial
model, then augment the model by setting 'User1 = 'User2. -}
--User('User1);
--User('User2);
--userType('User1) = 'MarketplaceVisitor;
--userType('User2) = 'ServiceVendor;

--------------------------------------------------
{- Query 7: What happens if we allow role subsumption? That is,
having permission to approve a file also gives you permission to
deny that file.
Note: Run the base set of axioms, then see if 'Erica can deny the
   UserManual
and the RatingEntry.-}
--mayApprove(u, f) => mayDeny(u,f);
--mayDeny(u,f) => User(u) & File(f);
--User('Erica) & userType('Erica) = 'ServiceVendor;
--File('UM) & docType('UM) = 'UserManual;
--File('RE) & docType('RE) = 'RatingEntry;
--------------------------------------------------
{- Query 8: What happens if we create a new docType, 'CustomerData?
Can any of the three user types open, write, or approve it?
Note: After adding the new docType, add it to the list of allowable
   docTypes.
Run the initial model, then try augmenting it with mayOpen, mayWrite, and
   mayApprove
for the 3 usertypes and the new docType.-}
--docType('CustomerData);
--User('MV) & userType('MV) = 'MarketplaceVisitor;
--User('SV) & userType('SV) = 'ServiceVendor;
--User('CS) & userType('CS) = 'CustomerService;
--File('f) & docType('f) = 'CustomerData;
```

B.3 Automatic Translation of RBAC.dl

```
{-RBAC.dl example of a company's permission system-}
{-modelled after the permission matrix on page 4 of http://citeseerx.ist.
   psu.edu/viewdoc/download;jsessionid=B1B59456CE9702EB37414FDFD98AEAB2?
   doi=10.1.1.157.9388&rep=rep1&type=pdf-}

{-mayWrite, mayOpen, mayApprove are roles-}

{-***MarketplaceVisitor***-}
{-mayOpen MarketingDocument, TermsDocument, RatingEntry; -}
{-mayWrite nothing;-}
{-mayApprove nothing;cannot-}
{-C_1 <=> (ForAll (Role mayOpen) (Or (Concept MarketingDocument) (Or (
   Concept TermsDocument) (Concept RatingEntry))))-}
MartketplaceVisitor(x) => C_1(x);
{-C_2 <=> (ForAll (Role mayOpen) (Or (Concept MarketingDocument) (Or (
   Concept TermsDocument) (Concept RatingEntry))))-}
C_1(x) & mayOpen(x, y2) => C_2(y2);
Truth => C_1(x) | exists y2.mayOpen(x, y2) & !C_2(y2);
C_2(x) => MarketingDocument(x) | TermsDocument(x) | RatingEntry(x);
MarketingDocument(x) => C_2(x);
TermsDocument(x) => C_2(x);
RatingEntry(x) => C_2(x);
{-C_4 <=> (ForAll (Role mayApprove) (Concept falsehood))-}
{-C_3 <=> (ForAll (Role mayWrite) (Concept falsehood))-}
C_4(x) & mayApprove(x, y4) => falsehood(y4);
Truth => C_4(x) | exists y4.mayApprove(x, y4) & !falsehood(y4);
MarketplaceVisitor(x) => C_3(x) & C_4(x);
C_3(x) & mayWrite(x, y6) => falsehood(y6);
Truth => C_3(x) | exists y6.mayWrite(x, y6) & !falsehood(y6);

{-***ServiceConsumer***-}
{-mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
   RatingEntry, TechInterface;-}
{-mayWrite RatingEntry;-}
{-mayApprove CustomerContract, TermsDocument;-}
{-C_5 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
   TermsDocument) (Or (Concept RatingEntry) (Concept TechInterface))))))
   -}
ServiceConsumer(x) => C_5(x);
{-C_6 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
   TermsDocument) (Or (Concept RatingEntry) (Concept TechInterface))))))
   -}
C_5(x) & mayOpen(x, y8) => C_6(y8);
```

```
Truth => C_5(x) | exists y8.mayOpen(x, y8) & !C_6(y8);
C_6(x) => UserManual(x) | MarketingDocument(x) | CustomerContract(x) |
   TermsDocument(x) | RatingEntry(x) | TechInterface(x);
UserManual(x) => C_6(x);
MarketingDocument(x) => C_6(x);
CustomerContract(x) => C_6(x);
TermsDocument(x) => C_6(x);
RatingEntry(x) => C_6(x);
TechInterface(x) => C_6(x);
{-C_7 <=> (ForAll (Role mayWrite) (Concept RatingEntry))-}
ServiceConsumer(x) => C_7(x);
C_7(x) & mayWrite(x, y10) => RatingEntry(y10);
Truth => C_7(x) | exists y10.mayWrite(x, y10) & !RatingEntry(y10);
{-C_8 <=> (ForAll (Role mayApprove) (Or (Concept CustomerContract) (
   Concept TermsDocument)))-}
ServiceConsumer(x) => C_8(x);
{-C_9 <=> (ForAll (Role mayApprove) (Or (Concept CustomerContract) (
   Concept TermsDocument)))-}
C_8(x) & mayApprove(x, y12) => C_9(y12);
Truth => C_8(x) | exists y12.mayApprove(x, y12) & !C_9(y12);
C_9(x) => CustomerContract(x) | TermsDocument(x);
CustomerContract(x) => C_9(x);
TermsDocument(x) => C_9(x);

{-***SoftwareEng***-}
{-mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   RatingEntry, TechInterface, DesignDoc;-}
{-mayWrite UserManual, TechInterface, InstallGuide, DesignDoc;-}
{-mayApprove MarketingDocument-}
{-C_10 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept RatingEntry) (Or (Concept TechInterface) (
   Concept DesignDoc))))))))-}
SoftwareEng(x) => C_10(x);
{-C_11 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept RatingEntry) (Or (Concept TechInterface) (
   Concept DesignDoc))))))))-}
C_10(x) & mayOpen(x, y14) => C_11(y14);
Truth => C_10(x) | exists y14.mayOpen(x, y14) & !C_11(y14);
C_11(x) => UserManual(x) | MarketingDocument(x) | TermsDocument(x) |
   InstallGuide(x) | RatingEntry(x) | TechInterface(x) | DesignDoc(x);
UserManual(x) => C_11(x);
MarketingDocument(x) => C_11(x);
TermsDocument(x) => C_11(x);
```

```
InstallGuide(x) => C_11(x);
RatingEntry(x) => C_11(x);
TechInterface(x) => C_11(x);
DesignDoc(x) => C_11(x);
{-C_12 <=> (ForAll (Role mayWrite) (Or (Concept UserManual) (Or (Concept
   TechInterface) (Or (Concept InstallGuide) (Concept DesignDoc)))))-}
SoftwareEng(x) => C_12(x);
{-C_13 <=> (ForAll (Role mayWrite) (Or (Concept UserManual) (Or (Concept
   TechInterface) (Or (Concept InstallGuide) (Concept DesignDoc)))))-}
C_12(x) & mayWrite(x, y16) => C_13(y16);
Truth => C_12(x) | exists y16.mayWrite(x, y16) & !C_13(y16);
C_13(x) => UserManual(x) | TechInterface(x) | InstallGuide(x) | DesignDoc(
   x);
UserManual(x) => C_13(x);
TechInterface(x) => C_13(x);
InstallGuide(x) => C_13(x);
DesignDoc(x) => C_13(x);
{-C_14 <=> (ForAll (Role mayApprove) (Concept MarketingDocument))-}
SoftwareEng(x) => C_14(x);
C_14(x) & mayApprove(x, y18) => MarketingDocument(y18);
Truth => C_14(x) | exists y18.mayApprove(x, y18) & !MarketingDocument(y18)
   ;

{-***ServiceVendor***-}
{-mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
   InstallGuide, TechInterface, DesignDoc, RatingEntry;-}
{-mayWrite nothing;-}
{-mayApprove UserManual, MarketingDocument, CustomerContract,
   TermsDocument, InstallGuide, TechInterface, DesignDoc;-}
{-C_15 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
   TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
   (Or (Concept DesignDoc) (Concept RatingEntry))))))))-}
ServiceVendor(x) => C_15(x);
{-C_16 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
   TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
   (Or (Concept DesignDoc) (Concept RatingEntry))))))))-}
C_15(x) & mayOpen(x, y20) => C_16(y20);
Truth => C_15(x) | exists y20.mayOpen(x, y20) & !C_16(y20);
C_16(x) => UserManual(x) | MarketingDocument(x) | CustomerContract(x) |
   TermsDocument(x) | InstallGuide(x) | TechInterface(x) | DesignDoc(x) |
    RatingEntry(x);
UserManual(x) => C_16(x);
MarketingDocument(x) => C_16(x);
```

```
CustomerContract(x) => C_16(x);
TermsDocument(x) => C_16(x);
InstallGuide(x) => C_16(x);
TechInterface(x) => C_16(x);
DesignDoc(x) => C_16(x);
RatingEntry(x) => C_16(x);
{-C_17 <=> (ForAll (Role mayWrite) (Concept falsehood))-}
ServiceVendor(x) => C_17(x);
C_17(x) & mayWrite(x, y22) => falsehood(y22);
Truth => C_17(x) | exists y22.mayWrite(x, y22) & !falsehood(y22);
{-C_18 <=> (ForAll (Role mayApprove) (Or (Concept UserManual) (Or (Concept
     MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
    TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
    (Concept DesignDoc))))))))-}
ServiceVendor(x) => C_18(x);
{-C_19 <=> (ForAll (Role mayApprove) (Or (Concept UserManual) (Or (Concept
     MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
    TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
    (Concept DesignDoc))))))))-}
C_18(x) & mayApprove(x, y24) => C_19(y24);
Truth => C_18(x) | exists y24.mayApprove(x, y24) & !C_19(y24);
C_19(x) => UserManual(x) | MarketingDocument(x) | CustomerContract(x) |
    TermsDocument(x) | InstallGuide(x) | TechInterface(x) | DesignDoc(x);
UserManual(x) => C_19(x);
MarketingDocument(x) => C_19(x);
CustomerContract(x) => C_19(x);
TermsDocument(x) => C_19(x);
InstallGuide(x) => C_19(x);
TechInterface(x) => C_19(x);
DesignDoc(x) => C_19(x);

{-***LegalDept***-}
{-mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
    InstallGuide, TechInterface, DesignDoc, RatingEntry;-}
{-mayWrite CustomerContract, TermsDocument;-}
{-mayApprove nothing;-}
{-C_20 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
    MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
    TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
    (Or (Concept DesignDoc) (Concept RatingEntry)))))))))-}
LegalDept(x) => C_20(x);
{-C_21 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
    MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
    TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
    (Or (Concept DesignDoc) (Concept RatingEntry)))))))))-}
```

```
C_20(x) & mayOpen(x, y26) => C_21(y26);
Truth => C_20(x) | exists y26.mayOpen(x, y26) & !C_21(y26);
C_21(x) => UserManual(x) | MarketingDocument(x) | CustomerContract(x) |
   TermsDocument(x) | InstallGuide(x) | TechInterface(x) | DesignDoc(x) |
    RatingEntry(x);
UserManual(x) => C_21(x);
MarketingDocument(x) => C_21(x);
CustomerContract(x) => C_21(x);
TermsDocument(x) => C_21(x);
InstallGuide(x) => C_21(x);
TechInterface(x) => C_21(x);
DesignDoc(x) => C_21(x);
RatingEntry(x) => C_21(x);
{-C_22 <=> (ForAll (Role mayWrite) (Or (Concept CustomerContract) (Concept
    TermsDocument)))-}
LegalDept(x) => C_22(x);
{-C_23 <=> (ForAll (Role mayWrite) (Or (Concept CustomerContract) (Concept
    TermsDocument)))-}
C_22(x) & mayWrite(x, y28) => C_23(y28);
Truth => C_22(x) | exists y28.mayWrite(x, y28) & !C_23(y28);
C_23(x) => CustomerContract(x) | TermsDocument(x);
CustomerContract(x) => C_23(x);
TermsDocument(x) => C_23(x);
{-C_24 <=> (ForAll (Role mayApprove) (Concept falsehood))-}
LegalDept(x) => C_24(x);
C_24(x) & mayApprove(x, y30) => falsehood(y30);
Truth => C_24(x) | exists y30.mayApprove(x, y30) & !falsehood(y30);

{-***ServiceProvider****-}
{-mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   TechInterface, RatingEntry;-}
{-mayWrite nothing;-}
{-mayApprove nothing;-}
{-C_25 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Concept RatingEntry))))))
   -}
ServiceProvider(x) => C_25(x);
{-C_26 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Concept RatingEntry))))))
   -}
C_25(x) & mayOpen(x, y32) => C_26(y32);
Truth => C_25(x) | exists y32.mayOpen(x, y32) & !C_26(y32);
```

```
C_26(x) => UserManual(x) | MarketingDocument(x) | TermsDocument(x) |
   InstallGuide(x) | TechInterface(x) | RatingEntry(x);
UserManual(x) => C_26(x);
MarketingDocument(x) => C_26(x);
TermsDocument(x) => C_26(x);
InstallGuide(x) => C_26(x);
TechInterface(x) => C_26(x);
RatingEntry(x) => C_26(x);
{-C_28 <=> (ForAll (Role mayApprove) (Concept falsehood))-}
{-C_27 <=> (ForAll (Role mayWrite) (Concept falsehood))-}
C_28(x) & mayApprove(x, y34) => falsehood(y34);
Truth => C_28(x) | exists y34.mayApprove(x, y34) & !falsehood(y34);
ServiceProvider(x) => C_27(x) & C_28(x);
C_27(x) & mayWrite(x, y36) => falsehood(y36);
Truth => C_27(x) | exists y36.mayWrite(x, y36) & !falsehood(y36);

{-***Marketing***-}
{-mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   TechInterface, DesignDoc, RatingEntry;-}
{-mayWrite MarketingDocument;-}
{-mayApprove nothing;-}
{-C_29 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Or (Concept DesignDoc) (
   Concept RatingEntry))))))))-}
Marketing(x) => C_29(x);
{-C_30 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Or (Concept DesignDoc) (
   Concept RatingEntry))))))))-}
C_29(x) & mayOpen(x, y38) => C_30(y38);
Truth => C_29(x) | exists y38.mayOpen(x, y38) & !C_30(y38);
C_30(x) => UserManual(x) | MarketingDocument(x) | TermsDocument(x) |
   InstallGuide(x) | TechInterface(x) | DesignDoc(x) | RatingEntry(x);
UserManual(x) => C_30(x);
MarketingDocument(x) => C_30(x);
TermsDocument(x) => C_30(x);
InstallGuide(x) => C_30(x);
TechInterface(x) => C_30(x);
DesignDoc(x) => C_30(x);
RatingEntry(x) => C_30(x);
{-C_31 <=> (ForAll (Role mayWrite) (Concept MarketingDocument))-}
Marketing(x) => C_31(x);
C_31(x) & mayWrite(x, y40) => MarketingDocument(y40);
Truth => C_31(x) | exists y40.mayWrite(x, y40) & !MarketingDocument(y40);
```

```
{-C_32 <=> (ForAll (Role mayApprove) (Concept falsehood))-}
Marketing(x) => C_32(x);
C_32(x) & mayApprove(x, y42) => falsehood(y42);
Truth => C_32(x) | exists y42.mayApprove(x, y42) & !falsehood(y42);

{-***TechEditor***-}
{-mayOpen UserManual, MarketingDocument, TermsDocument, InstallGuide,
   TechInterface, DesignDoc, RatingEntry;-}
{-mayWrite UserManual, InstallGuide, TechInterface, DesginDoc;-}
{-mayApprove nothing;-}
{-C_33 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Or (Concept DesignDoc) (
   Concept RatingEntry))))))))-}
TechEditor(x) => C_33(x);
{-C_34 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept TermsDocument) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Or (Concept DesignDoc) (
   Concept RatingEntry))))))))-}
C_33(x) & mayOpen(x, y44) => C_34(y44);
Truth => C_33(x) | exists y44.mayOpen(x, y44) & !C_34(y44);
C_34(x) => UserManual(x) | MarketingDocument(x) | TermsDocument(x) |
   InstallGuide(x) | TechInterface(x) | DesignDoc(x) | RatingEntry(x);
UserManual(x) => C_34(x);
MarketingDocument(x) => C_34(x);
TermsDocument(x) => C_34(x);
InstallGuide(x) => C_34(x);
TechInterface(x) => C_34(x);
DesignDoc(x) => C_34(x);
RatingEntry(x) => C_34(x);
{-C_35 <=> (ForAll (Role mayWrite) (Or (Concept UserManual) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Concept DesginDoc)))))-}
TechEditor(x) => C_35(x);
{-C_36 <=> (ForAll (Role mayWrite) (Or (Concept UserManual) (Or (Concept
   InstallGuide) (Or (Concept TechInterface) (Concept DesginDoc)))))-}
C_35(x) & mayWrite(x, y46) => C_36(y46);
Truth => C_35(x) | exists y46.mayWrite(x, y46) & !C_36(y46);
C_36(x) => UserManual(x) | InstallGuide(x) | TechInterface(x) | DesginDoc(
   x);
UserManual(x) => C_36(x);
InstallGuide(x) => C_36(x);
TechInterface(x) => C_36(x);
DesginDoc(x) => C_36(x);
{-C_37 <=> (ForAll (Role mayApprove) (Concept falsehood))-}
TechEditor(x) => C_37(x);
```

```
C_37(x) & mayApprove(x, y48) => falsehood(y48);
Truth => C_37(x) | exists y48.mayApprove(x, y48) & !falsehood(y48);

{-***CustomerService***-}
{-mayOpen UserManual, MarketingDocument, CustomerContract, TermsDocument,
   InstallGuide, TechInterface, DesignDoc, RatingEntry;-}
{-mayWrite CustomerContract-}
{-mayApprove nothing;-}
{-C_38 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
   TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
   (Or (Concept DesignDoc) (Concept RatingEntry)))))))))-}
CustomerService(x) => C_38(x);
{-C_39 <=> (ForAll (Role mayOpen) (Or (Concept UserManual) (Or (Concept
   MarketingDocument) (Or (Concept CustomerContract) (Or (Concept
   TermsDocument) (Or (Concept InstallGuide) (Or (Concept TechInterface)
   (Or (Concept DesignDoc) (Concept RatingEntry)))))))))-}
C_38(x) & mayOpen(x, y50) => C_39(y50);
Truth => C_38(x) | exists y50.mayOpen(x, y50) & !C_39(y50);
C_39(x) => UserManual(x) | MarketingDocument(x) | CustomerContract(x) |
   TermsDocument(x) | InstallGuide(x) | TechInterface(x) | DesignDoc(x) |
    RatingEntry(x);
UserManual(x) => C_39(x);
MarketingDocument(x) => C_39(x);
CustomerContract(x) => C_39(x);
TermsDocument(x) => C_39(x);
InstallGuide(x) => C_39(x);
TechInterface(x) => C_39(x);
DesignDoc(x) => C_39(x);
RatingEntry(x) => C_39(x);
{-C_40 <=> (ForAll (Role mayWrite) (Concept CustomerContract))-}
CustomerService(x) => C_40(x);
C_40(x) & mayWrite(x, y52) => CustomerContract(y52);
Truth => C_40(x) | exists y52.mayWrite(x, y52) & !CustomerContract(y52);
{-C_41 <=> (ForAll (Role mayApprove) (Concept falsehood))-}
CustomerService(x) => C_41(x);
C_41(x) & mayApprove(x, y54) => falsehood(y54);
Truth => C_41(x) | exists y54.mayApprove(x, y54) & !falsehood(y54);
```

Appendix C

Translation Examples

C.1 Basic DL Example

Description Logic:

```
\\Carnivore Ontology
Carnivore ≡ ∀eatsFood.Meat
```

Description Logic Formalism:

Code C.1: Carnivore DL Formalism

```
\\Carnivore Ontology
Carnivore definedas forall eatsFood.Meat
```

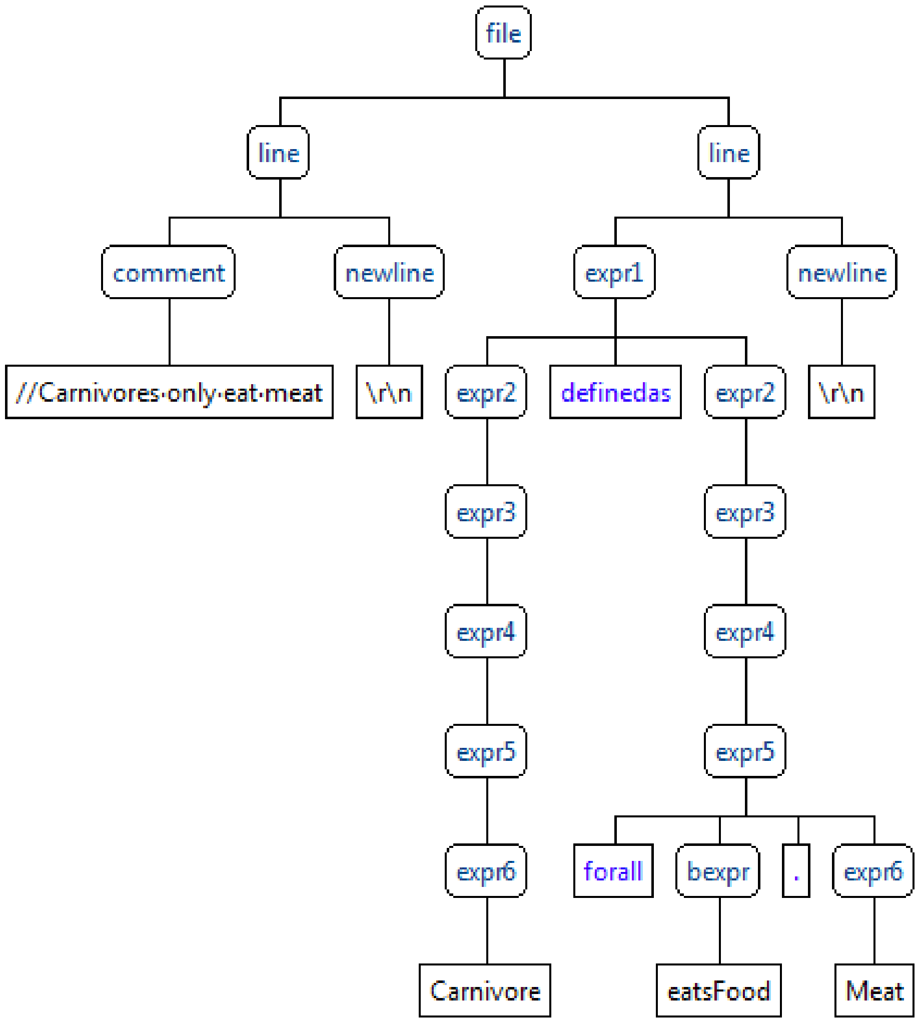


Figure C.1: Abstract Syntax Tree of the Carnivore Ontology

Razor Input:

Code C.2: Carnivore GL Translation

```
{-Carnivore Ontology-}
Carn(x) & eatsFood(x,y) => Meat(y);
Truth => Carn(x) | exists y.eatsFood(x,y) & NotMeat(y);
Meat(y) & NotMeat(y) => Falsehood;
```

C.2 Food DL Example

Food Description Logic Formalism:

Code C.3: Food DL Formalism

```
//Food Ontology

//A food item is either meat, fruit, vegetable, or starch
Food => Meat | Vegetable | Fruit | Starch

//Vegetarians do not eat meat
Vegetarian definedas forall eatsFood.(! Meat)

//Carnivores only eat meat
Carnivore definedas forall eatsFood.Meat

//If a food is a vegetable and a starch, it's a potato
Vegetable & Starch => Potato

//A food cannot be a vegetable and a fruit
Vegetable & Fruit => Falsehood

//Some people have a favorite food
exists faveFood.Truth

//Dads like meat and potatos
Dad => forall faveFood.(Meat & Potato)

//A balanced dinner has meat, vegetables, and starch
BalancedDinner definedas forall mealContains.(Meat & Vegetable & Starch)

//People that don't like fruits or vegetables are unhealthy
! forall likesFood.(Fruit | Vegetable) => UnhealthyPerson

//Kids don't like vegetables
Kid => forall likesFood.(!Vegetable)

//A restaurant is a place that serves food
Restaurant definedas exists servesFood.Truth

//An Italian restaurant is a restaurant that serves Italian food
ItalianRestaurant => Restaurant & forall servesFood.Italian

//A steakhouse is a restaurant that serves meat
Steakhouse => Restaurant & forall servesFood.Meat
```

Food Razor Input:

Code C.4: Food GL Translation

```
{-Food Ontology-}

{-A food item is either meat, fruit, vegetable, or starch-}
Food(x) => Meat(x) | Vegetable(x) | Fruit(x) | Starch(x);

{-Vegetarians do not eat meat-}
{-C_1 <=> (ForAll (Role eatsFood) (Not (Concept Meat)))-}
Vegetarian(x) & eatsFood(x, y2) => C_1(y2);
Truth => Vegetarian(x) | exists y2.eatsFood(x, y2) & !C_1(y2);
C_1(x) => NegatedMeat(x);
Meat(x) & NegatedMeat(x) => Falsehood;
Truth => C_1(x);

{-Carnivores only eat meat-}
Carnivore(x) & eatsFood(x, y4) => Meat(y4);
Truth => Carnivore(x) | exists y4.eatsFood(x, y4) & !Meat(y4);

{-If a food is a vegetable and a starch, it's a potato-}
Vegetable(x) & Starch(x) => Potato(x);

{-A food cannot be a vegetable and a fruit-}
Vegetable(x) & Fruit(x) => Falsehood;

{-Some people have a favorite food-}
Truth => exists y5.faveFood(x, y5) & Truth;

{-Dads like meat and potatos-}
{-C_2 <=> (ForAll (Role faveFood) (And (Concept Meat) (Concept Potato)))-}
Dad(x) => C_2(x);
{-C_3 <=> (ForAll (Role faveFood) (And (Concept Meat) (Concept Potato)))-}
C_2(x) & faveFood(x, y7) => C_3(y7);
Truth => C_2(x) | exists y7.faveFood(x, y7) & !C_3(y7);
C_3(x) => Meat(x) & Potato(x);
Meat(x) & Potato(x) => C_3(x);

{-A balanced dinner has meat, vegetables, and starch-}
{-C_4 <=> (ForAll (Role mealContains) (And (Concept Meat) (And (Concept
   Vegetable) (Concept Starch))))-}
BalancedDinner(x) & mealContains(x, y9) => C_4(y9);
Truth => BalancedDinner(x) | exists y9.mealContains(x, y9) & !C_4(y9);
C_4(x) => Meat(x) & Vegetable(x) & Starch(x);
Meat(x) & Vegetable(x) & Starch(x) => C_4(x);
```

```
{-People that don't like fruits or vegetables are unhealthy-}
exists y10.likesFood(x, y10) & Truth => UnhealthyPerson(x);

{-Kids don't like vegetables-}
{-C_5 <=> (ForAll (Role likesFood) (Not (Concept Vegetable)))-}
Kid(x) => C_5(x);
{-C_6 <=> (ForAll (Role likesFood) (Not (Concept Vegetable)))-}
C_5(x) & likesFood(x, y12) => C_6(y12);
Truth => C_5(x) | exists y12.likesFood(x, y12) & !C_6(y12);
C_6(x) => NegatedVegetable(x);
Vegetable(x) & NegatedVegetable(x) => Falsehood;
Truth => C_6(x);

{-A restaurant is a place that serves food-}
Restaurant(x) => exists y14.servesFood(x, y14) & Truth;
exists y14.servesFood(x, y14) & Truth => Restaurant(x);

{-An Italian restaurant is a restaurant that serves Italian food-}
{-C_7 <=> (ForAll (Role servesFood) (Concept Italian))-}
ItalianRestaurant(x) => Restaurant(x) & C_7(x);
C_7(x) & servesFood(x, y16) => Italian(y16);
Truth => C_7(x) | exists y16.servesFood(x, y16) & !Italian(y16);

{-A steakhouse is a restaurant that serves meat-}
{-C_8 <=> (ForAll (Role servesFood) (Concept Meat))-}
Steakhouse(x) => Restaurant(x) & C_8(x);
C_8(x) & servesFood(x, y18) => Meat(y18);
Truth => C_8(x) | exists y18.servesFood(x, y18) & !Meat(y18);
```

C.3 Atom OWL Example

Atom OWL Formalism, Large parts ommited since otherwise this single file would be 25 pages long.

Code C.5: Atom Initial Formalism

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://ontology.dumontierlab.com/atom-complex-disjoint#"
     xml:base="http://ontology.dumontierlab.com/atom-complex-disjoint"
     xmlns:dc="http://purl.org/dc/elements/1.1/"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:ontology="http://ontology.dumontierlab.com/">
    <owl:Ontology rdf:about="http://ontology.dumontierlab.com/atom-complex
        -disjoint">
        <owl:versionInfo rdf:datatype="http://www.w3.org/2001/XMLSchema#
            string">2.0</owl:versionInfo>
        <dc:date rdf:datatype="http://www.w3.org/2001/XMLSchema#date
            ">2008-02-01</dc:date>
        <dc:title rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            Atom Ontology (disjoint)</dc:title>
        <dc:description rdf:datatype="http://www.w3.org/2001/XMLSchema#
            string">Disjoint union of atom types.</dc:description>
        <dc:publisher rdf:datatype="http://www.w3.org/2001/XMLSchema#
            string">Dumontier Lab</dc:publisher>
        <dc:creator rdf:datatype="http://www.w3.org/2001/XMLSchema#string
            ">Michel Dumontier</dc:creator>
        <dc:format rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
            application/rdf+xml</dc:format>
        <dc:language rdf:datatype="http://www.w3.org/2001/XMLSchema#string
            ">en</dc:language>
        <owl:imports rdf:resource="http://ontology.dumontierlab.com/atom-
            primitive"/>
    </owl:Ontology>



    <!--
    ///////////////////////////////////////////////////////////////////////////////////////

    //
    // Annotation properties
    //
```

```
///////////////////////////////////////////////////////////////////////////////////////

 -->

<owl:AnnotationProperty rdf:about="http://www.w3.org/2002/07/owl#
   versionInfo"/>
<owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/
   creator"/>
<owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/
   publisher"/>
<owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/
   language"/>
<owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/
   date"/>
<owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/
   description"/>
<owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/
   format"/>
<owl:AnnotationProperty rdf:about="http://purl.org/dc/elements/1.1/
   title"/>



<!--
///////////////////////////////////////////////////////////////////////////////////////

//
// Classes
//
///////////////////////////////////////////////////////////////////////////////////////

 -->




<!-- http://ontology.dumontierlab.com/ActiniumAtom -->

<owl:Class rdf:about="http://ontology.dumontierlab.com/ActiniumAtom"/>



<!-- http://ontology.dumontierlab.com/AluminiumAtom -->
```

```
    <owl:Class rdf:about="http://ontology.dumontierlab.com/AluminiumAtom
       "/>

  ...


<!-- Generated by the OWL API (version 3.0.0.1450) http://owlapi.
   sourceforge.net -->
```

Atom DL Formalism:

Code C.6: Atom DL Translation

```
ArsenicAtom => Class
BerkeliumAtom => Class
HydrogenAtom => Class
PromethiumAtom => Class
YtterbiumAtom => Class
UraniumAtom => Class
EinsteiniumAtom => Class
NeptuniumAtom => Class
TitaniumAtom => Class
UnunhexiumAtom => Class
TechnetiumAtom => Class
CalciumAtom => Class
BromineAtom => Class
LeadAtom => Class
KryptonAtom => Class
RutherfordiumAtom => Class
RhodiumAtom => Class
AmericiumAtom => Class
GermaniumAtom => Class
HassiumAtom => Class
PhosphorusAtom => Class
PoloniumAtom => Class
UnunseptiumAtom => Class
ChromiumAtom => Class
CuriumAtom => Class
MolybdenumAtom => Class
ThoriumAtom => Class
StrontiumAtom => Class
BismuthAtom => Class
IodineAtom => Class
RadiumAtom => Class
ZincAtom => Class
FranciumAtom => Class
EuropiumAtom => Class
```

```
OsmiumAtom => Class
NiobiumAtom => Class
UnunoctiumAtom => Class
TerbiumAtom => Class
CarbonAtom => Class
CaesiumAtom => Class
LutetiumAtom => Class
MeitneriumAtom => Class
ScandiumAtom => Class
SiliconAtom => Class
ArgonAtom => Class
BariumAtom => Class
HafniumAtom => Class
HolmiumAtom => Class
PraseodymiumAtom => Class
XenonAtom => Class
UnununiumAtom => Class
DysprosiumAtom => Class
NeonAtom => Class
TinAtom => Class
UnunbiumAtom => Class
TantalumAtom => Class
BoronAtom => Class
LawrenciumAtom => Class
IronAtom => Class
RutheniumAtom => Class
RheniumAtom => Class
AluminiumAtom => Class
GalliumAtom => Class
FluorineAtom => Class
PalladiumAtom => Class
PlutoniumAtom => Class
UnunquadiumAtom => Class
ChlorineAtom => Class
CopperAtom => Class
ManganeseAtom => Class
MercuryAtom => Class
ThalliumAtom => Class
SodiumAtom => Class
BerylliumAtom => Class
IndiumAtom => Class
ProtactiniumAtom => Class
YttriumAtom => Class
ErbiumAtom => Class
NobeliumAtom => Class
```

```
NickelAtom => Class
TungstenAtom => Class
UnunniliumAtom => Class
TelluriumAtom => Class
CaliforniumAtom => Class
CadmiumAtom => Class
LithiumAtom => Class
LanthanumAtom => Class
SamariumAtom => Class
SeleniumAtom => Class
AntimonyAtom => Class
AstatineAtom => Class
GoldAtom => Class
HeliumAtom => Class
PlatinumAtom => Class
PotassiumAtom => Class
VanadiumAtom => Class
UnuntriumAtom => Class
DubniumAtom => Class
NeodymiumAtom => Class
ThuliumAtom => Class
SulfurAtom => Class
BohriumAtom => Class
IridiumAtom => Class
RubidiumAtom => Class
RadonAtom => Class
ZirconiumAtom => Class
ActiniumAtom => Class
GadoliniumAtom => Class
FermiumAtom => Class
OxygenAtom => Class
NitrogenAtom => Class
UnunpentiumAtom => Class
CeriumAtom => Class
CobaltAtom => Class
MagnesiumAtom => Class
MendeleviumAtom => Class
SeaborgiumAtom => Class
SilverAtom => Class
```

Translated Razor Input for Atom:

Code C.7: Atom GF Translation

```
ArsenicAtom(x) => Class(x);
BerkeliumAtom(x) => Class(x);
HydrogenAtom(x) => Class(x);
```

```
PromethiumAtom(x) => Class(x);
YtterbiumAtom(x) => Class(x);
UraniumAtom(x) => Class(x);
EinsteiniumAtom(x) => Class(x);
NeptuniumAtom(x) => Class(x);
TitaniumAtom(x) => Class(x);
UnunhexiumAtom(x) => Class(x);
TechnetiumAtom(x) => Class(x);
CalciumAtom(x) => Class(x);
BromineAtom(x) => Class(x);
LeadAtom(x) => Class(x);
KryptonAtom(x) => Class(x);
RutherfordiumAtom(x) => Class(x);
RhodiumAtom(x) => Class(x);
AmericiumAtom(x) => Class(x);
GermaniumAtom(x) => Class(x);
HassiumAtom(x) => Class(x);
PhosphorusAtom(x) => Class(x);
PoloniumAtom(x) => Class(x);
UnunseptiumAtom(x) => Class(x);
ChromiumAtom(x) => Class(x);
CuriumAtom(x) => Class(x);
MolybdenumAtom(x) => Class(x);
ThoriumAtom(x) => Class(x);
StrontiumAtom(x) => Class(x);
BismuthAtom(x) => Class(x);
IodineAtom(x) => Class(x);
RadiumAtom(x) => Class(x);
ZincAtom(x) => Class(x);
FranciumAtom(x) => Class(x);
EuropiumAtom(x) => Class(x);
OsmiumAtom(x) => Class(x);
NiobiumAtom(x) => Class(x);
UnunoctiumAtom(x) => Class(x);
TerbiumAtom(x) => Class(x);
CarbonAtom(x) => Class(x);
CaesiumAtom(x) => Class(x);
LutetiumAtom(x) => Class(x);
MeitneriumAtom(x) => Class(x);
ScandiumAtom(x) => Class(x);
SiliconAtom(x) => Class(x);
ArgonAtom(x) => Class(x);
BariumAtom(x) => Class(x);
HafniumAtom(x) => Class(x);
HolmiumAtom(x) => Class(x);
```

```
PraseodymiumAtom(x) => Class(x);
XenonAtom(x) => Class(x);
UnununiumAtom(x) => Class(x);
DysprosiumAtom(x) => Class(x);
NeonAtom(x) => Class(x);
TinAtom(x) => Class(x);
UnunbiumAtom(x) => Class(x);
TantalumAtom(x) => Class(x);
BoronAtom(x) => Class(x);
LawrenciumAtom(x) => Class(x);
IronAtom(x) => Class(x);
RutheniumAtom(x) => Class(x);
RheniumAtom(x) => Class(x);
AluminiumAtom(x) => Class(x);
GalliumAtom(x) => Class(x);
FluorineAtom(x) => Class(x);
PalladiumAtom(x) => Class(x);
PlutoniumAtom(x) => Class(x);
UnunquadiumAtom(x) => Class(x);
ChlorineAtom(x) => Class(x);
CopperAtom(x) => Class(x);
ManganeseAtom(x) => Class(x);
MercuryAtom(x) => Class(x);
ThalliumAtom(x) => Class(x);
SodiumAtom(x) => Class(x);
BerylliumAtom(x) => Class(x);
IndiumAtom(x) => Class(x);
ProtactiniumAtom(x) => Class(x);
YttriumAtom(x) => Class(x);
ErbiumAtom(x) => Class(x);
NobeliumAtom(x) => Class(x);
NickelAtom(x) => Class(x);
TungstenAtom(x) => Class(x);
UnunniliumAtom(x) => Class(x);
TelluriumAtom(x) => Class(x);
CaliforniumAtom(x) => Class(x);
CadmiumAtom(x) => Class(x);
LithiumAtom(x) => Class(x);
LanthanumAtom(x) => Class(x);
SamariumAtom(x) => Class(x);
SeleniumAtom(x) => Class(x);
AntimonyAtom(x) => Class(x);
AstatineAtom(x) => Class(x);
GoldAtom(x) => Class(x);
HeliumAtom(x) => Class(x);
```

```
PlatinumAtom(x) => Class(x);
PotassiumAtom(x) => Class(x);
VanadiumAtom(x) => Class(x);
UnuntriumAtom(x) => Class(x);
DubniumAtom(x) => Class(x);
NeodymiumAtom(x) => Class(x);
ThuliumAtom(x) => Class(x);
SulfurAtom(x) => Class(x);
BohriumAtom(x) => Class(x);
IridiumAtom(x) => Class(x);
RubidiumAtom(x) => Class(x);
RadonAtom(x) => Class(x);
ZirconiumAtom(x) => Class(x);
ActiniumAtom(x) => Class(x);
GadoliniumAtom(x) => Class(x);
FermiumAtom(x) => Class(x);
OxygenAtom(x) => Class(x);
NitrogenAtom(x) => Class(x);
UnunpentiumAtom(x) => Class(x);
CeriumAtom(x) => Class(x);
CobaltAtom(x) => Class(x);
MagnesiumAtom(x) => Class(x);
MendeleviumAtom(x) => Class(x);
SeaborgiumAtom(x) => Class(x);
SilverAtom(x) => Class(x);
```

Appendix D

Razor Queries and Models

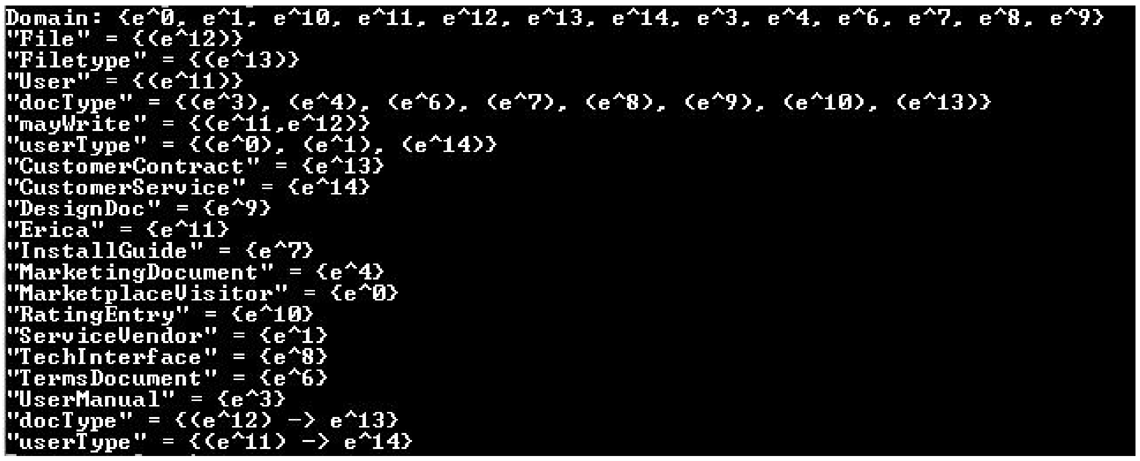
Many sample Razor queries and their are provided in Section 5.3. Additional queries and their corresponding Razor models are provided here. These queries are run using the hand-translated“default deny” and “default allow” perspectives of the RBAC example which are reproduced in Section B.1 and Section B.2, respectively.

D.1 Additional User Type Query

The following query examines user types in the RBAC policy example. It asks which user types a user may take on given that they may write a Customer Contract. The query is shown in D.1 and the resulting Razor model is shown in Figure D.1.

Code D.1: User Type Query

```
{-Query: If Erica writes a CustomerContract, what type of user is she?-}
--User('Erica);
--exists f . mayWrite('Erica, f) & docType(f) = 'CustomerContract;
--exists ut . userType('Erica) = ut;
```



```
Domain: {e^0, e^1, e^10, e^11, e^12, e^13, e^14, e^3, e^4, e^6, e^7, e^8, e^9}
"File" = {(e^12)}
"Filetype" = {(e^13)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^6), (e^7), (e^8), (e^9), (e^10), (e^13)}
"mayWrite" = {(e^11,e^12)}
"userType" = {(e^0), (e^1), (e^14)}
"CustomerContract" = {e^13}
"CustomerService" = {e^14}
"DesignDoc" = {e^9}
"Erica" = {e^11}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^12) -> e^13}
"userType" = {(e^11) -> e^14}
```

Figure D.1: Erica could be a Customer Service Employee

Razor shows that, given that the user may write a Customer Contract, they may be a Customer Service Employee. When Razor is asked for the next minimal model, we are told that no more exist. Razor correctly points out that Customer Service Employees are the only user type that may write Customer Contracts. Note that the same results are obtained with both the "default deny" and "default allow" perspectives.

D.2 Additional File Type Query

The following query examines how each user type may interact with a new file type. The new file type, called "Customer Data", has been added to the list of acceptable file types with the following axioms:

Code D.2: Adding a New File Type

```
docType('CustomerData);
docType(ft) => ft = 'UserManual | ft = 'MarketingDocument | ft = '
   CustomerContract | ft = 'TermsDocument
| ft = 'InstallGuide | ft = 'TechInterface | ft = 'DesignDoc | ft = '
   RatingEntry | ft = 'CustomerData;
'CustomerData = 'UserManual => Falsehood;
'CustomerData = 'MarketingDocument => Falsehood;
'CustomerData = 'CustomerContract => Falsehood;
'CustomerData = 'TermsDocument => Falsehood;
'CustomerData = 'InstallGuide => Falsehood;
'CustomerData = 'TechInterface => Falsehood;
'CustomerData = 'DesignDoc => Falsehood;
'CustomerData = 'RatingEntry => Falsehood;
```

After adding those axioms, we can then ask Razor to display models where the user types may open, write, and approve a Customer Data file. In queries D.3, D.4, and D.5 Razor is asked to display models where a Service Vendor may open, write, and approve a Customer Data file, respectively.

Code D.3: Opening a Customer Data File (Default Deny)

```
{- Query: What happens if we create a new docType, 'CustomerData?
Can a Service Vendor open, write, or approve it? -}
User('SV) & userType('SV) = 'ServiceVendor;
File('f) & docType('f) = 'CustomerData;
mayOpen('SV, 'f);
```

Code D.4: Writing a Customer Data File (Default Deny)

```
{- Query: What happens if we create a new docType, 'CustomerData?
Can a Service Vendor open, write, or approve it? -}
User('SV) & userType('SV) = 'ServiceVendor;
File('f) & docType('f) = 'CustomerData;
mayWrite('SV, 'f);
```

Code D.5: Approving a Customer Data File (Default Deny)

```
{- Query: What happens if we create a new docType, 'CustomerData?
Can a Service Vendor open, write, or approve it? -}
User('SV) & userType('SV) = 'ServiceVendor;
File('f) & docType('f) = 'CustomerData;
mayApprove('SV, 'f);
```

After running each these queries, Razor states that no models exists for any of these queries. This behavior is expected, since the "default deny" perspective enumerates only the documents a user may access. Since the new Customer Data file was not added to the whitelist of any of the user types, a Service Vendor cannot access the new file type in any way. Note that this result holds for both of the other two user types.

Now, we can run the above queries in the "default allow" perspective. We ask if the Service Vendor may open, write, and approve a Customer Data file in queries D.6, D.7 and D.8, respectively.

Code D.6: Opening a Customer Data File (Default Allow)

```
{- Query: What happens if we create a new docType, 'CustomerData?
Can a Service Vendor open, write, or approve it? -}
User('SV) & userType('SV) = 'ServiceVendor;
File('f) & docType('f) = 'CustomerData;
mayOpen('SV, 'f);
```

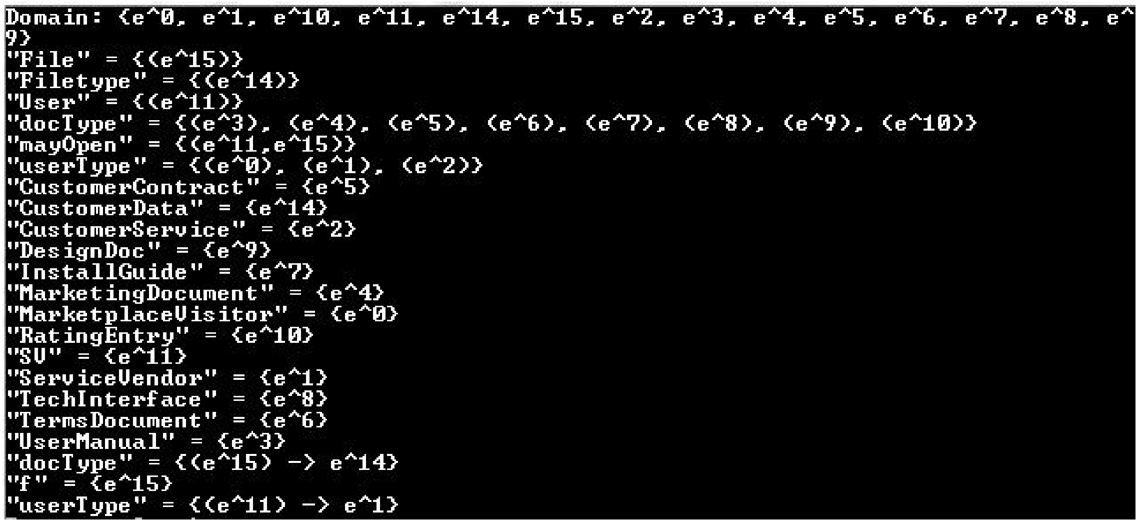
Code D.7: Writing a Customer Data File (Default Allow)

```
{- Query: What happens if we create a new docType, 'CustomerData?
Can a Service Vendor open, write, or approve it? -}
User('SV) & userType('SV) = 'ServiceVendor;
File('f) & docType('f) = 'CustomerData;
mayWrite('SV, 'f);
```

Code D.8: Approving a Customer Data File (Default Allow)

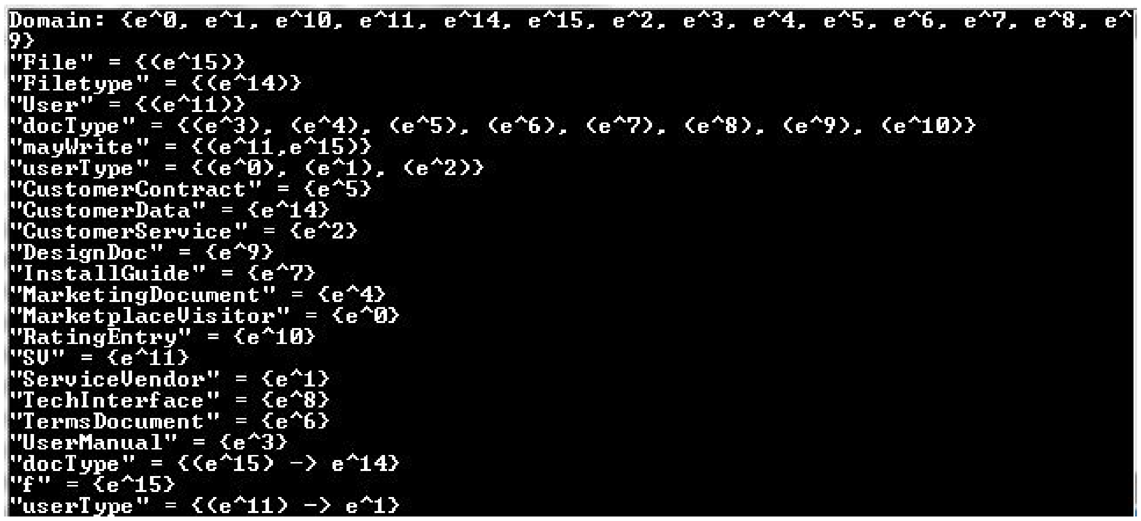
```
{- Query: What happens if we create a new docType, 'CustomerData?
Can a Service Vendor open, write, or approve it? -}
User('SV) & userType('SV) = 'ServiceVendor;
File('f) & docType('f) = 'CustomerData;
mayApprove('SV, 'f);
```

Unlike the "default deny" queries, Razor produces models for each of the above queries. The models corresponding to D.6, D.7, and D.8 are presented in Figures D.2, D.3, and D.4, respectively.



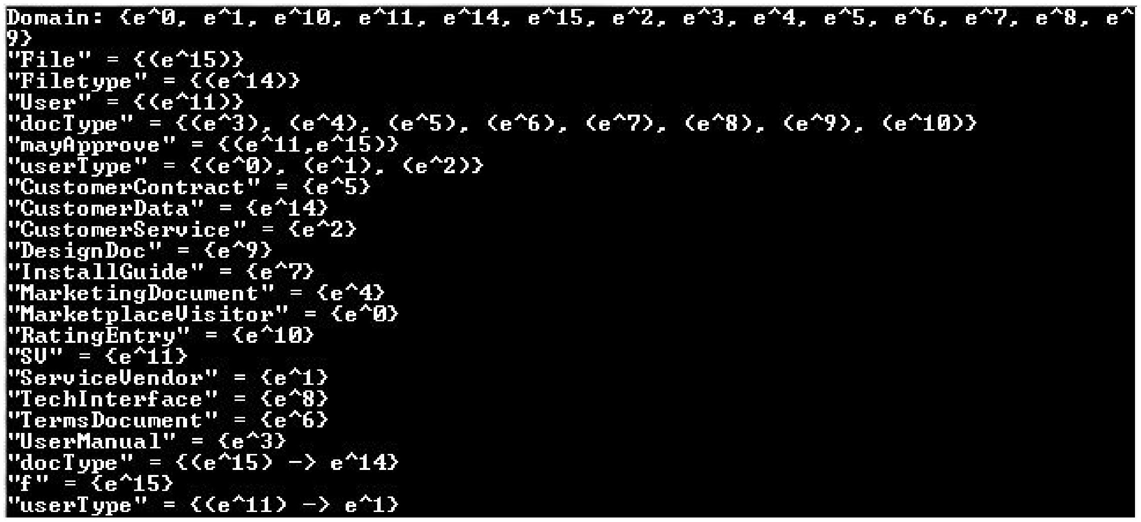
```
Domain: {e^0, e^1, e^10, e^11, e^14, e^15, e^2, e^3, e^4, e^5, e^6, e^7, e^8, e^
9}
"File" = {(e^15)}
"Filetype" = {(e^14)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^5), (e^6), (e^7), (e^8), (e^9), (e^10)}
"mayOpen" = {(e^11,e^15)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerData" = {e^14}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"SV" = {e^11}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^15) -> e^14}
"f" = {e^15}
"userType" = {(e^11) -> e^1}
```

Figure D.2: Erica can open a Customer Data File



```
Domain: {e^0, e^1, e^10, e^11, e^14, e^15, e^2, e^3, e^4, e^5, e^6, e^7, e^8, e^
9}
"File" = {(e^15)}
"Filetype" = {(e^14)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^5), (e^6), (e^7), (e^8), (e^9), (e^10)}
"mayWrite" = {(e^11,e^15)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerData" = {e^14}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"SV" = {e^11}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^15) -> e^14}
"f" = {e^15}
"userType" = {(e^11) -> e^1}
```

Figure D.3: Erica can write a Customer Data File



```
Domain: {e^0, e^1, e^10, e^11, e^14, e^15, e^2, e^3, e^4, e^5, e^6, e^7, e^8, e^
9}
"File" = {(e^15)}
"Filetype" = {(e^14)}
"User" = {(e^11)}
"docType" = {(e^3), (e^4), (e^5), (e^6), (e^7), (e^8), (e^9), (e^10)}
"mayApprove" = {(e^11,e^15)}
"userType" = {(e^0), (e^1), (e^2)}
"CustomerContract" = {e^5}
"CustomerData" = {e^14}
"CustomerService" = {e^2}
"DesignDoc" = {e^9}
"InstallGuide" = {e^7}
"MarketingDocument" = {e^4}
"MarketplaceVisitor" = {e^0}
"RatingEntry" = {e^10}
"SV" = {e^11}
"ServiceVendor" = {e^1}
"TechInterface" = {e^8}
"TermsDocument" = {e^6}
"UserManual" = {e^3}
"docType" = {(e^15) -> e^14}
"f" = {e^15}
"userType" = {(e^11) -> e^1}
```

Figure D.4: Erica can approve a Customer Data File

Because the “default allow” perspective enumerates only the files that user types may not interact with, adding the new file type without specifying access control axioms allows every user type (not just Service Vendors) to perform every action type on it.