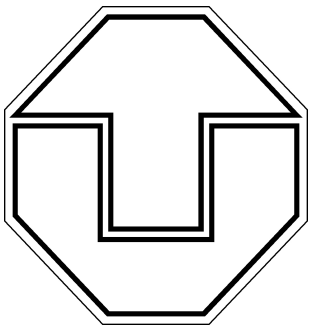
$\mathcal{ALC}$-LTL Formula Generator



M. Fareed Arif

International Center for Computational Logic (ICCL)

Technische Universität Dresden

Institut für Theoretische Informatik

Master in Computational Logic

yet to be decided

I would like to dedicate this thesis to my loving parents ...

Acknowledgements

Und ich möchte allen in Deutschland danken, dass sie mir geholfen haben dieses Wissen zu erlangen.

Abstract

$\mathcal{ALC}$-LTL is a temporalized extension of the Description Logic $\mathcal{ALC}$. A program that generates $\mathcal{ALC}$-LTL formulae can be used to test developed applications in the area of temporalized Description Logics (DLs). In this report, we present algorithms and the implementation detail about such a program which randomly generates $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae. The randomness in this program is controlled using a uniformed probability distribution. The program has several parameters which control the generation of $\mathcal{ALC}$-concept descriptions, $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae. For the generation of different formulae, we have recursive algorithms which are accompanied by its termination and soundness proves. The program implement these algorithm using the OWL API which provides a support to access, manipulate and store $\mathcal{ALC}$-concept descriptions, $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae.

Contents

1 Introduction **1**

2 Preliminaries **3**

2.1 The Description Logic $\mathcal{ALC}$ . . . 3

2.1.1 Syntax . . . 3

2.1.2 Semantics . . . 4

2.2 Temporalizing Description Logic $\mathcal{ALC}$-LTL . . . 5

2.2.1 Syntax . . . 5

2.2.2 Semantics . . . 6

2.2.3 Rigid Roles & Concepts . . . 7

2.3 Web Ontology Language (OWL) . . . 7

2.3.1 The OWL API . . . 8

2.3.2 $\mathcal{ALC}$ vs OWL API Correspondence . . . 8

3 Generating Algorithms **10**

3.1 $\mathcal{ALC}$-Concept Descriptions Algorithm . . . 10

3.2 $\mathcal{ALC}$-axioms Algorithm . . . 15

3.3 $\mathcal{ALC}$-LTL formulae Algorithm . . . 17

3.4 Incorporating Rigid Roles & Concepts . . . 22

4 Implementation & Results **23**

4.1 Implementation . . . 23

4.2 Program Options . . . 26

4.3 Results . . . 26

5 Conclusions **30**

5.1 Future Directions 30

A Appendix **31**

References **36**

Chapter 1

Introduction

Temporal logic such as linear temporal logic (LTL) Pnueli (1977) combined with the description logic $\mathcal{ALC}$ is presented in Baader et al. (2008a). This temporalized description logic called $\mathcal{ALC}$-LTL can represent dynamic aspects in many application domains. To motivate this fact Baader et al. (2008a) states some real world examples. In one such example, a person name BOB carry a head injury but does not loose his consciousness between concussion and examination is stated using an $\mathcal{ALC}$-LTL formula. A $\mathcal{ALC}$-LTL formulae consist a set of $\mathcal{ALC}$-axioms with the temporal operators (i.e., negation, conjunction, disjunction, next and Until) in front. An $\mathcal{ALC}$-axiom is either an $\mathcal{ALC}$-assertions or a GCI (General Concept Inclusion) contructed using $\mathcal{ALC}$-concept descriptions.

In this report, we explain the implementation detail of a program which generates $\mathcal{ALC}$-LTL formulae. To generate a $\mathcal{ALC}$-LTL formula, we first requiress more basic construct which is an $\mathcal{ALC}$-axiom. Both $\mathcal{ALC}$-assertions and GCIs are called $\mathcal{ALC}$-axioms. The program generates $\mathcal{ALC}$-axioms using set of concept names N_C, set of role names N_R and set of individual names N_I. If N_C, N_R and N_I are not provided then program also generates these sets randomly. The program is given with several parameters. These parameters help the end user to control the length and type of generated formulae as well as to provide access and storage information. The randomness in the program is controlled using a uniformed probability distribution which ensures this fact that all symbols and operators occurred in a generated formula have an equal chance of selection. In this report, we present three recursive algorithms which were implemented in the end. The pseudocode

of these generating algorithms is accompanied by their soundness and termination proves. The program generates $\mathcal{ALC}$-concept descriptions, $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae. of desired lengths. The OWL-API is used for accessing, manipulating and storing the generated $\mathcal{ALC}$-axioms. In order to access, manipulate and store $\mathcal{ALC}$-LTL formulea, we use the OWL-API along with some additional data structures in order to accommodate the temporal conditions.

In Chapter 2, we provide preliminaries about the formalisms which are used in later chapters. The Chapter 3 explain the pseudocode of recursive algorithms which are used to generate $\mathcal{ALC}$-concept descriptions, $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae. In Chapter 4, the implementation of the program is explained. This report also contains a short conclusion with some future directions for our work.

Chapter 2

Preliminaries

2.1 The Description Logic $\mathcal{ALC}$

In this section, we present the syntax and semantics of the description logic $\mathcal{ALC}$ as it is treated in Baader et al. (2008b).

2.1.1 Syntax

Definition 1 (Baader et al. (2008b))**.** Let N_C is a set of *concept names*, N_R is a set of *role names* and N_I is a set *individual names*. The set of $\mathcal{ALC}$ *concept descriptions* is the smallest set satisfying the following properties:

- Every *concept name*, $\top$ and $\bot$ are $\mathcal{ALC}$-concept descriptions,
- If C, D are $\mathcal{ALC}$ concept descriptions, $r \in N_R$, then the following are $\mathcal{ALC}$ concept descriptions:

$$\begin{aligned} \neg\, C &\quad \text{(Complement)} \\ C \sqcap D &\quad \text{(Conjunction)} \\ C \sqcup D &\quad \text{(Disjunction)} \\ \exists r.C &\quad \text{(Existential restriction)} \\ \forall r.C &\quad \text{(Value restriction)} \end{aligned}$$

Definition 2 (Baader et al. (2008b))**.** [General Concept Inclusion (GCI)] A *general concept inclusion (GCI)* is of form $C \sqsubseteq D$ where C, D are $\mathcal{ALC}$-concept

descriptions.

A finite set of General Concept Inclusions (GCIs) is called a TBox which defines the background knowledge of an application domain.

Definition 3. Baader et al. (2008b)[Assertion] An *assertion* is of the form $a : C$ or $(a, b) : r$ where C is an $\mathcal{ALC}$-concept description, r is a role name, and a, b are individual names.

An ABox is a finite set of assertions which describe an application domain at any instance. *We call both, GCIs and Assertions, $\mathcal{ALC}$-axioms.*

2.1.2 Semantics

For the semantics of $\mathcal{ALC}$, we introduce the notion of *interpretation.*

Definition 4. Baader et al. (2008b) An $\mathcal{ALC}$ interpretation $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ is a non-empty set, called the domain of $\mathcal{I}$, and a mapping $\cdot^{\mathcal{I}}$ that assigns:

- a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ to each concept name $A \in N_C$,
- $\top = \Delta^{\mathcal{I}}$ and $\bot = \emptyset$.
- an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ to each individual name $a \in N_I$ and
- a binary relation $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each role name $r \in N_R$,

The interpretation of complex concepts is then defined as follows:

- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$,
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$,
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$,
- $(\exists r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} |$ there is a $y \in \Delta^{\mathcal{I}}$ with $(x, y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$,
- $(\forall r.C)^{\mathcal{I}} = \{x \in \Delta^{\mathcal{I}} |$ for all $y \in \Delta^{\mathcal{I}}, (x, y) \in r^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\}$,
- An $\mathcal{ALC}$ interpretation $\mathcal{I}$ is a model of a GCI $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$,

- An $\mathcal{ALC}$ interpretation $\mathcal{I}$ is a model of a TBox $\mathcal{T}$ if it is a model of every GCI in $\mathcal{T}$.
- An $\mathcal{ALC}$ interpretation $\mathcal{I}$ is a model of an assertion axiom $a : C$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$ and
- An $\mathcal{ALC}$ interpretation $\mathcal{I}$ is a model of an assertion axiom $(a, b) : r$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$.
- An $\mathcal{ALC}$ interpretation $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ if it is a model of every axiom in $\mathcal{A}$.

In the next section, we define the temporal extension of the description logic $\mathcal{ALC}$.

2.2 Temporalizing Description Logic $\mathcal{ALC}$-LTL

Temporal logic such as linear temporal logic (LTL) Pnueli (1977) combined with the description logic $\mathcal{ALC}$ can represent dynamic aspects in a number of application domains. To motivate this fact Lutz et al. (2008) and Baader et al. (2008a) states some useful examples in which temporal aspects are playing an important role in DLs. For example, if we want to state that the German football team will eventually becomes FIFA World Cup champions then an $\mathcal{ALC}$-LTL formula can represent it in the following way:

$$\Diamond(\exists winner.FIFA\_WORLD\_CUP\_2014) \sqcap Germany : FOOT\_BALL\_TEAMS$$

In this section, we present the $\mathcal{ALC}$-LTL syntax in which the temporal operators are only allowed to occur in front of $\mathcal{ALC}$-axioms but not inside the concept descriptions as it is described in Baader et al. (2008a).

2.2.1 Syntax

Definition 5. Baader et al. (2008a) $\mathcal{ALC}$-LTL formulae are inductively defined as follows:

- If α is an $\mathcal{ALC}$-axiom then α is an $\mathcal{ALC}$-LTL formula;

- If ϕ, ψ are $\mathcal{ALC}$-LTL formulas, then the following are $\mathcal{ALC}$-LTL formulas:

$$\begin{aligned} \neg\ \phi \quad & \text{(Negation)} \\ \phi \wedge \psi \quad & \text{(Conjunction)} \\ \phi \vee \psi \quad & \text{(Disjunction)} \\ \phi\ \mathsf{U}\ \psi \quad & \text{(Until)} \\ \mathsf{X}\ \psi \quad & \text{(Next)} \end{aligned}$$

2.2.2 Semantics

Definition 6. Baader et al. (2008a) An *$\mathcal{ALC}$-LTL* structure is a *sequence* $\mathfrak{I} = (\mathcal{I}_i)_{i=0,1,\ldots}$ of $\mathcal{ALC}$ interpretations $\mathcal{I}_i = (\Delta, .^{\mathcal{I}_i})$ such that $a^{\mathcal{I}_i} = a^{\mathcal{I}_j}$ for all individual names a and for all $i, j \in \{0, 1, 2, \ldots, \}$. Given an $\mathcal{ALC}$-LTL formula ϕ, an $\mathcal{ALC}$-LTL structure $\mathfrak{I} = (\mathcal{I}_i)_{i=0,1,\ldots,}$ and a time point $i \in \{0, 1, 2, \ldots\}$, validity of ϕ in $\mathfrak{I}$ at time i (written $\mathfrak{I}, i \models \phi$) is defined inductively:

- $\mathfrak{I}, i \models C \sqsubseteq D$ iff $C^{\mathcal{I}_i} \subseteq D^{\mathcal{I}_i}$,
- $\mathfrak{I}, i \models a : C$ iff $a^{\mathcal{I}_i} \in C^{\mathcal{I}_i}$,
- $\mathfrak{I}, i \models (a, b) : r$ iff $(a^{\mathcal{I}_i}, b^{\mathcal{I}_i}) \in r^{\mathcal{I}_i}$,
- $\mathfrak{I}, i \models \phi \wedge \psi$ iff $\mathfrak{I}, i \models \phi$ and $\mathfrak{I}, i \models \psi$,
- $\mathfrak{I}, i \models \phi \vee \psi$ iff $\mathfrak{I}, i \models \phi$ or $\mathfrak{I}, i \models \psi$,
- $\mathfrak{I}, i \models \neg\ \phi$ iff $\mathfrak{I}, i \not\models \phi$,
- $\mathfrak{I}, i \models \mathsf{X}\ \phi$ iff $\mathfrak{I}, i + 1 \models \phi$,
- $\mathfrak{I}, i \models \phi\ \mathsf{U}\ \psi$ iff there is $k \geq i$ such that $\mathfrak{I}, k \models \psi$ and $\mathfrak{I}, j \models \phi$ for all j, $i \leq j < k$.

To improve *$\mathcal{ALC}$-LTL* formula readability, we use following abbreviations:

- true for $\phi \vee \neg\phi$ which describes that $\phi \vee \neg\phi$ is satisfiable at present,

- $\Diamond\phi$ for $\mathsf{true}\ \mathsf{U}\ \phi$ which describes that ϕ will become satisfiable sometime in future and we use $\Box\phi$ for $\neg(\ \mathsf{true}\ \mathsf{U}\ \neg\phi)$ which describes that ϕ remains satisfiable always in future.

Now, we furter move on to explain rigid roles and rigid concepts as it was explained in Baader et al. (2008a). A concept is called *rigid concept* if its interpretation does not change over time.

Definition 7. Baader et al. (2008a) For any given $\mathcal{ALC}$-LTL strcture $\mathfrak{I} = (\mathcal{I}_i)_{i=0,1,\dots}$ respects a rigid concept name iff $C^{(\mathcal{I}_i)} = C^{(\mathcal{I}_j)}$ holds for all $i, j \in \{0, 1, 2, \dots\}$ and C is a rigid concept name.

A role is called *rigid role* if its interpretation does not change over time.

Definition 8. Baader et al. (2008a) For any given $\mathcal{ALC}$-LTL strcture $\mathfrak{I} = (\mathcal{I}_i)_{i=0,1,\dots}$ respects a rigid role name iff $r^{(\mathcal{I}_i)} = r^{(\mathcal{I}_j)}$ holds for all $i, j \in \{0, 1, 2, \dots\}$ and r is a rigid role name.

2.2.3 Rigid Roles & Concepts

In some situation, we require to have rigid roles and rigid concept names beacuse it rejects the unintended models thus helping to draw only useful information from a knowledge base as it is stated in Baader et al. (2008a) with an elaborate example which enforce a rule of nature that a human being and his ancestory will remain the same over his life-time. In Baader et al. (2008a), it is also stated that the reasoning becomes harder in the presence of rigid roles.

2.3 Web Ontology Language (OWL)

In K. Smith et al. (2004), it is described that Web Ontology Language (OWL) is a family of knowledge representation languages for authoring ontologies. An OWL ontology represents a domain using set of concepts and the relationships among these concepts. An OWL ontology make it possible to do different type of reasoning (i.e., subsumption, satisfiability and equivalence checking etc.) in a domain.

2.3.1 The OWL API

The OWL API is described in Bechhofer et al. (2003) which states that it is a programming interface to access and manipulate OWL ontologies. In order to use this API for our project, we need to understand its corresponds with the description logic $\mathcal{ALC}$. Here, we elaborate a correspondence which exist between the description logic $\mathcal{ALC}$ and OWL API.

2.3.2 $\mathcal{ALC}$ vs OWL API Correspondence

The OWL API corresponds with description logic $\mathcal{ALC}$ in such a way that the $\mathcal{ALC}$-concept descriptions are of type OWLClassExpression having following objects, relations and restrictions:

- Objects:
  - An individual in set N_I corresponds to an object of type OWLIndividual in OWL API.
  - An $\mathcal{ALC}$-concept corresponds to a class of type OWLClass in OWL API which is a collection sharing some common characteristics.
  - A role in set N_R corresponds to a property of OWLObjectProperty in OWL API which describe a binary relationships among individuals.
- Relations:
  - *Conjunction* of $\mathcal{ALC}$-concept descriptions is of type OWLObjectIntersectionOf in OWL API,
  - *Disjunction* of $\mathcal{ALC}$-concept descriptions is of type OWLObjectUnionOf in OWL API,
  - *Complement* of an $\mathcal{ALC}$-concept description is of type OWLObjectComplementOf in OWL API.
- Restrictions:
  - *Existential restriction* in $\mathcal{ALC}$-concept descriptions is of type OWLObjectSomeValuesFrom.

– *Value restriction* $\mathcal{ALC}$-concept descriptions is of type OWLObjectSomeValuesFrom.

Chapter 3

Generating Algorithms

In this chapter, we define algorithms that generate $\mathcal{ALC}$-concept descriptions, $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae according to their inductive syntactic definitions explained in the preceding chapter. These algorithms are defined recursively in such a way that only well-formed $\mathcal{ALC}$-concept descriptions, $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae are generated. These algorithms have several parameters in place to control the length of generated output (i.e., concept descriptions, axioms and formulae), to exclude tautologies from the output and in the end store it in an ontology file for later access.

3.1 $\mathcal{ALC}$-Concept Descriptions Algorithm

In this section, we define a recursive algorithm to generate $\mathcal{ALC}$-concept descriptions. The length of the generated concept descriptions is always bounded by an input parameter which is amounting to the occurrences of randomly selected operators occur in it. The length of $\mathcal{ALC}$-concept description is defined as follows:

Definition 9. For an arbitrary $\mathcal{ALC}$-concept description, its the length $|\_|$ is computed inductively:

- $|\alpha| := 1$ where α is a concept name (i.e., $\alpha \in N_C$),
- $|\neg\alpha| := |\alpha| + 1$ where α is an $\mathcal{ALC}$-concept description.

- $|\alpha_1 \sqcap \alpha_2| := |\alpha_1| + |\alpha_2| + 1$ and $|\alpha_1 \sqcup \alpha_2| := |\alpha_1| + |\alpha_2| + 1$ where α_1, α_2 are $\mathcal{ALC}$-concept descriptions,
- $|\exists r.\alpha| = |\alpha| + 1$ and $|\forall r.\alpha| = |\alpha| + 1$ where r is a role name $r \in N_R$ and α is an $\mathcal{ALC}$-concept description.

In our algorithm, we define a uniform probability distribution for the random selection of every operator or a symbol from a set. This uniform selection helps to restrict the frequency of every operator that could occur in an $\mathcal{ALC}$-concept description. Thus, in a generated $\mathcal{ALC}$-concept description all the operators and symbols occur with an equal number of chance.

Pseudocode

The $\mathcal{ALC}$-concept description algorithm takes an input argument n of type integer denoting the desired number of operators that should exist in the generated concept descriptions. This algorithm returns a generated $\mathcal{ALC}$-concept description of provided length n.

- ```
  function alc_cd (n: Integer, apply_negation: Boolean)
  begin
    (i)1
    ...
  end
  ```

  1. The input argument n is a positive integer number which specify the length of the generated concept description that is to be returned in the end.
  2. To restrict the the multiple consecutive occurrences of negation in the generated $\mathcal{ALC}$-concept description, we use the *apply_negation* Boolean flag.

- (i)
  ```

```
...
  if n = 1 then begin
     concept_name := randomly choose a symbol from set N_C;
     return concept_description(concept_name);
  else if n > 1 then begin
     op := Randomly choose a symbol from set {¬, ⊔, ⊓, ∃, ∀};
     (ii)2...
  end if
...
```

1. If given length n is *1* then a unit concept description is generated from a randomly chosen with uniform distribution, a concept name from the set of concepts N_C,
2. If given length n is greater than unit length then algorithm chooses an operator *op* with uniform distribution from the set $\{\neg, \sqcup, \sqcap, \exists, \forall\}$ and then recursively apply it.

- (ii)

```
...
    switch (op)
      case op == ¬ begin
        if apply_negation == true then
          φ := alc_cd(n, true);
          return φ;
        else
          φ := alc_cd(n-1, true);
          return complement_conception_description(φ);
        end if
      end case
      case op == ⊓ begin
        n := n - 1;
        length_1 = randomly select a number less then n;
        length_2 = n - length_1;
```

```
      φ := alc_cd(length_1, false);
      ψ := alc_cd(length_2, false);
      return  conjunctive_concept_description(φ,ψ);
    end case
    case op == ⊔:
      n := n - 1;
      length_1 = randomly select a number less then n;
      length_2 = n - length_1;
      φ := alc_cd(length_1, false);
      ψ := alc_cd(length_2, false);
      return  disjunctive_concept_description(φ,ψ);
    end case
    case op == ∃:
      role = randomly choose a role from set N_R;
       ψ := alc_cd(n-1, false);
      return  ext_rest_concept_description(role,ψ);
    end case
    case op == ∀:
      role = randomly choose a role from set N_R;
      ψ := alc_cd(n-1, false);
      return  value_rest_concept_description(role ,ψ);
    end case
  end switch
...
```

1. In order to avoid producing only symmetrical $\mathcal{ALC}$-concept descriptions in case of $\phi \sqcap \psi, \phi \sqcup \psi$, our algorithm perform a random partition the length n into *length_1* and *length_2*.
2. Objects in $\mathcal{ALC}$ concept descriptions are represented in the following way:
   - complement_conception_description(ϕ) correspond to the complement of an $\mathcal{ALC}$-concept description (i.e., $\neg\phi$),
   - conjunctive_concept_description(ϕ,ψ) correspond to conjunction in $\mathcal{ALC}$-concept description (i.e., $\phi \sqcap \psi$),

- disjunctive_concept_description(ϕ,ψ) correspond to disjunction in $\mathcal{ALC}$-concept description (i.e., $\phi \sqcup \psi$),
- ext_rest_concept_description(role,ψ) correspond to existential restriction in $\mathcal{ALC}$-concept description (i.e., $\exists role.\phi$),
- value_rest_concept_description(role,ψ) correspond to value restriction in $\mathcal{ALC}$-concept description (i.e., $\forall role.\phi$),

Above algorithm holds the following properties:

Proposition 10. The procedure $alc\_cd(n : Integer, negation_flag : Boolean)$ always terminates.

Proof. Suppose algorithm runs for a pair of values $\langle n, false\rangle$ supplied as an input argument where $n \in \mathbb{N}$ is an arbitrary positive integer and negation flag is set to false. In each iteration of algorithm, the first element in this pair n is decrement by 1. Now assume our procedure does not terminate. In that case, we an infinite decreasing sequence ($\langle n, false\rangle \rightarrow \langle n-1, false\rangle \rightarrow \langle n-2, false\rangle \rightarrow \dots$) where " $\rightarrow$ " represents a one step iteration in our procedure. But again in next iteration But it never occurs because the iterations are bounded by the length of n and our procedure always terminates when n becomes 1. In a case where our procedure selects to apply two consecutive negations the flag is flipped to restrict this case and n is not decremented. Due to that fact that every operator which is applied is uniformly selected therefore applying consecutive negations can occur only finite number of times during any run and does not effect the termination of our procedure. Hence, our algorithm always terminates. □

Proposition 11. The generated $\mathcal{ALC}$-concept description by our procedure is always a well-formed $\mathcal{ALC}$-concept description.

Proof. To prove it, we proceed by doing induction on the size of an $\mathcal{ALC}$-concept description which is returned by our procedure when it terminates. Let α be an arbitrary $\mathcal{ALC}$-concept description returned by our procedure then:

- case($\alpha = C$): Clearly, the input size n is 1 and our procedure simply returns a well-formed $\mathcal{ALC}$-concept generation by returning a concept name selected from set of concepts N_C. By definition of $\mathcal{ALC}$-concept description, we know that every concept name is also an $\mathcal{ALC}$-concept description.

- case($\alpha = C \sqcap D$): In this case the size n is greater than 1. Our procedure decrement n by 1 and select to apply negation operation to an already well-formed $\mathcal{ALC}$-concept description. According to the syntactic definition of $\mathcal{ALC}$-concept description, the resulting formula is again a well-formed $\mathcal{ALC}$-concept description.

- case($\alpha = C \sqcup D$): Similar to the previous case.

- case($\alpha = \exists role.\beta$): In this case again the size n is greater than 1 and our procedure selects to do an existential restriction on an already generated $\mathcal{ALC}$-concept description β. By definition again it returns a well-formed $\mathcal{ALC}$-concept description.

- case($\alpha = \forall role.\beta$): Similar to the previous case.

Hence, every generated formula by our algorithm is a well-formed $\mathcal{ALC}$-LTL formula. □

3.2 $\mathcal{ALC}$-axioms Algorithm

An $\mathcal{ALC}$-axiom is defined to be either a GCI(General Concept Inclusion) or an assertion where assertion can be further categorized as concept assertion, role assertion or negated role assertion. Our following algorithm use a uniform distribution to decide which type of $\mathcal{ALC}$-axiom should be generated. The input argument m in our algorithm denotes the length of an $\mathcal{ALC}$-concept description. The algorithm returns a generated $\mathcal{ALC}$-axiom.

Pseudocode

- ```
  function alc_axiom(m : Integer)
    begin
    op := Randomly choose a symbol from set
       {CONCEPT_ASSERTION,ROLE_ASSERTION,NEG_ROLE_ASSERTION,⊑}
    (i)
    ...
  end
  ```
  ```

1. The input parameter m restrict the length of an $\mathcal{ALC}$-concept description and only useful in a case when the algorithm decides to generate a concept assertion axiom.

- (i)

```
...
  switch (op) begin
     case op == CONCEPT_ASSERTION begin
        α := randomly choose an individual from set N_I
        φ := alc_cd(m, false);
        return ConceptAssertionAxiom(α,φ);
     end case
     case op == ROLE_ASSERTION begin
        α := randomly choose an individual from set N_I
        β  := randomly choose an individual from set N_I
        role = randomly choose a role from set N_R
        return RoleAssertionAxiom(α,β,role);
     end case
     case op == NEG_ROLE_ASSERTION begin
        α := randomly choose an individual from set N_I
        β  := randomly choose an individual from set N_I
        role = randomly choose a complement role from set N_R
        return NegativeRoleAssertionAxiom(α,β,role);
    end case
    case op == ⊑ begin
        m := m - 1;
        length_1 = randomly select a number less then n;
        length_2 = n - length_1;
        φ := alc_cd(length_1);
        ψ := alc_cd(length_2);
        return GCIAxiom(φ, φ);
    end case
  end switch
...
```

1. In case of concept assertion axiom generation, we call the procedure $alc\_cd(m, false)$ to obtain a concept description of a given length m.

2. In case of a GCI(General Concept Inclusion) axiom generation, the $alc\_cd(m)$ is called twice to obtain two concept descriptions of different lengths namely as *length_1* and *length_2* which sum-up to the input argument m.

3. As described in preceding chapter, $\mathcal{ALC}$-axioms has following types:

   - ConceptAssertionAxiom(α,ϕ) correspond with concept assertion axiom.
   - RoleAssertionAxiom(α,β,role) is type for role assertion axiom.
   - NegativeRoleAssertionAxiom(α,β,role) represents negative role assertion axiom.
   - GCIAxiom(ϕ, ϕ) is a GCI (General Concept Inclusion) assertion axiom.

3.3 $\mathcal{ALC}$-LTL formulae Algorithm

Now, we define an algorithm that generates $\mathcal{ALC}$-LTL formulae using recursion thus characterizing the inductive syntactic definition of $\mathcal{ALC}$-LTL explained in the preceding chapter.
The length of an $\mathcal{ALC}$-LTL formula is formally defined as follows:

Definition 12 (Syntactic Length Computation)**.** For an arbitrary $\mathcal{ALC}$-LTL, the length is inductively computed in the following way:

- $|\varphi| := 0$ where ϕ is an $\mathcal{ALC}$-LTL atomic formula,
- $\neg\varphi| := |\varphi| + 1$ and $|X\varphi| := |\varphi| + 1$ where φ is an $\mathcal{ALC}$-LTL formula,
- $|\varphi \wedge \psi| := |\varphi| + |\psi| + 1$, $|\varphi \vee \psi| := |\varphi| + |\psi| + 1$ and $|\varphi \mathsf{U} \psi| := |\varphi| + |\psi| + 1$ where φ and ψ are $\mathcal{ALC}$-LTL formulae,

Pseudocode

The input argument m in this algorithm denotes the length of $\mathcal{ALC}$-axioms occur in an $\mathcal{ALC}$-LTL formula and and argument n bound the length of $\mathcal{ALC}$-LTL formula.

- `function alc_ltl_Formula (m: Integer, n: Integer): LTLOWLAxiom`
  `begin`
  `(i)`
  ...
  `end`

  1. The input arguments m is a positive integer number which specify the length of $\mathcal{ALC}$-axioms occurred in an $\mathcal{ALC}$-LTL formula generated in the end,
  2. The input arguments n is a positive integer number which specify the length of $\mathcal{ALC}$-LTL formula that should be returned by our procedure.

- (i)

  ...
  `if n = 1 then begin`
  ϕ `:= alc_axiom(m, false);`
  `return AtomicAxiom(`ϕ`);`
  `else if n > 1 then begin`
  `op := Randomly select an symbol from` $\{\neg,\ X, \vee, \wedge, \mathsf{U}\}$;
  `(ii)`
  ...
  `end if`
  ...

  1. $\mathcal{ALC}$-LTL atomic formula is an $\mathcal{ALC}$-axiom.
  2. If given length n is 1 then an atomic $\mathcal{ALC}$-LTL formula is generated by calling $\mathcal{ALC}$-axiom generation procedure with in put argument m to obtain an atomic assertion axiom as specified earlier.

3. If given length n is greater than unit length then algorithm uniformly select an operator from set $\{\neg, X, \vee, \wedge, \mathsf{U}\}$ and then recursively apply it.

- (ii)

```
...
   switch (op) begin
      case op == ¬ begin
        φ := alc_ltl_Formula(m, n-1);
        return NegativeFormula(φ);
      end case
      case op == X begin
        φ := alc_ltl_Formula(n-1);
        return NextFormula(φ);
      end case
      case op == ∨ begin
        n := n - 1;
        length_1 = randomly select a number less then n;
        length_2 = n - length_1;
        φ := alc_ltl_Formula(m,length_1);
        ψ := alc_ltl_Formula(m,length_2);
        return ConjunctiveFormula(φ,ψ);
      end case
      case op == ∧ begin
        n := n - 1;
        length_1 = randomly select a number less then n;
        length_2 = n - length_1;
        φ := alc_ltl_Formula(m, length_1);
        ψ := alc_ltl_Formula(m, length_2);
        return ConjunctiveFormula(φ,∧,ψ);
      end case
      case op == U begin
        n := n - 1;
```

```
        length_1 = randomly select a number less then n;
        length_2 = n - length_1;
        φ := alc_ltl_Formula(m, length_1);
        ψ := alc_ltl_Formula(m, length_2);
        return new UntilFormula(φ,U,ψ);
      end case
    end switch
  end if
...
```

1. For $\mathcal{ALC}$-LTL formulae, the following types of constructs are defined:

   - AtomicAxiom(ϕ) correspond to the construction of an atomic $\mathcal{ALC}$-LTL formula,
   - NegativeFormula(ϕ) correspond to a negative $\mathcal{ALC}$-LTL formula (i.e., $\neg\varphi$),
   - NextFormula(ϕ) represent a formula of form $X\phi$ in $\mathcal{ALC}$-LTL,
   - ConjunctiveFormula(ϕ,ψ) is of form $\phi \wedge \psi$ in $\mathcal{ALC}$-LTL,
   - DisjunctiveFormula(ϕ,ψ) correspond to an $\mathcal{ALC}$-LTL formula of form $\phi \vee \psi$,
   - UntilFormula(ϕ,ψ) correspond to an $\mathcal{ALC}$-LTL formula of form $\phi \mathsf{U} \psi$,

The following properties holds by $\mathcal{ALC}$-LTL formulae generating algorithm:

Proposition 13. The procedure $alc\_ltl\_Formula(m : Integer, n : Integer)$ always terminates.

Proof. Suppose algorithm runs for a pair of values $\langle m, n\rangle$ supplied as an input arguments where $m, n \in \mathbb{N}$ are arbitrary positive integers. In each iteration of algorithm, the first element in this pair n is decrement by 1 where as the element m remains constant through the complete run. Now assume our procedure does not terminate. In that case, we an infinite decreasing sequence ($\langle n, m\rangle \rightarrow \langle n-1, m\rangle \rightarrow \langle n-2, m\rangle \rightarrow \dots$) where " $\rightarrow$ " represents a one step iteration in our procedure. But it never occurs because the iterations are bounded by the length of n and

our procedure always terminates when n becomes 1. In our procedure, we call an $\mathcal{ALC}$-axiom generator to generate axioms of length m which is already shown to terminate and returns a well-formed $\mathcal{ALC}$-axiom. Hence, our algorithm always terminates. □

Proposition 14. Any generated formula is always a well-formed $\mathcal{ALC}$-LTL formula.

Proof. The generated $\mathcal{ALC}$-LTL formula returned by our algorithm is always a well-formed $\mathcal{ALC}$-LTL formula. To prove it, we proceed by doing induction on the size of an $\mathcal{ALC}$-LTL formula generated by our procedure. Let ϕ be an arbitrary $\mathcal{ALC}$-LTL formula returned by our procedure then:

- $\mathsf{case}(\phi = \alpha)$: Clearly, the input size n is 1 and our procedure simply returns a well-formed $\mathcal{ALC}$-axiom of size m. By the syntactic definition of $\mathcal{ALC}$-LTL formulae, we know that every $\mathcal{ALC}$-axiom is also an $\mathcal{ALC}$-LTL formula.
- $\mathsf{case}(\phi = \neg\varphi)$: In this case the size n is greater than 1. Our procedure decrement n by 1 and select to apply negation operation to an already well-formed formula φ. According to the syntactic definition of $\mathcal{ALC}$-LTL formulae, the resulting formula is again a well-formed $\mathcal{ALC}$-LTL formula.
- $\mathsf{case}(\phi = X\phi)$: Similar to the previous case.
- $\mathsf{case}(\phi = \varphi \wedge \psi)$: In this case again the size n is greater than 1 and our procedure decides to apply conjunction to an already generated $\mathcal{ALC}$-LTL formulae φ and ψ. By definition, again it returns a well-formed $\mathcal{ALC}$-LTL formula.
- $\mathsf{case}(\phi = \varphi \vee \psi)$: Similar to the previous case.
- $\mathsf{case}(\phi = \varphi \mathsf{U} \psi)$: Similar to the previous case.

Hence, every generated formula by our algorithm is a well-formed $\mathcal{ALC}$-LTL formula. □

3.4 Incorporating Rigid Roles & Concepts

The motivation to use rigid concepts and roles is already explained in the prevoius chapter. To incorporate rigid concept and rigid role, we shall require to do the following tasks.

1. First, we need to maintain a separate set of rigid concept names and rigid role names. For this purpose, we need to include a set G_C which contains only rigid concept names. We also need to include another set G_R which contains only rigid role names. These sets G_C, G_R are mutually exclusive to the set of concepts N_C and roles N_R.

2. We need to introduce an algorithm which generates $\mathcal{ALC}$-axioms that contains rigid concepts and roles.

3. While generating $\mathcal{ALC}$-LTL formulae, the algorithm needs to select $\mathcal{ALC}$-axioms that include rigid concepts and roles.

4. In the end we also may require a separate ontology file to store and later on access these rigid roles and concepts.

Chapter 4

Implementation & Results

In this chapter, we explain the implementation of our program and its usage.

4.1 Implementation

The $\mathcal{ALC}$-LTL generator is implemented using the language JAVA. Our program generate set of concepts N_C, set of roles N_R, set of individuals N_I. These generated sets are further used in the generation of $\mathcal{ALC}$-axioms. We stored and later on access these sets and axioms using the standard OWL API. For generating $\mathcal{ALC}$-LTL formulae, we had included some data structures in OWL API and then used it to generate the required $\mathcal{ALC}$-LTL formulae. The details about our program is explained in this section.

System Architecture

Our program is a collection of JAVA classes which can be categorized in two major groups. The first group contains only core classes which are the implementations of the pseudocode algorithms presented in the previous chapter. The second group is called utility classes which provide utility function to the core classes. The details about core classes as well as utility classes is as follows.

Core Classes

Following are the core classes in our program.

1. VocGenerator performs the following tasks:
   (a) It generates a set of concepts names, set of role names and set of Individual names of provided length,
   (b) It stores the generated sets using StoreOntology utility class which is explained later.
2. EprGenerator performs the the following tasks:
   (a) Generate $\mathcal{ALC}$-concept descriptions,
   (b) It uses LoadOntology to access the sets which were generated by VocGenerator.
3. AxiomGenerator does the following tasks:
   (a) It generates $\mathcal{ALC}$-concept assertion axioms.
   (b) It generates $\mathcal{ALC}$-role assertion axioms.
   (c) It generates $\mathcal{ALC}$-negated role assertion axioms.
   (d) It generates GCIs (General Concept Inclusions).
   (e) It uses LoadOntology to access the the set of concepts names, role names and individual names and EprGenerator class to generator $\mathcal{ALC}$-concept descriptions of provided length.
4. ALCGenerator does the following tasks:
   (a) It uses AxiomGenerator obtain $\mathcal{ALC}$-axioms.
   (b) It store the generated $\mathcal{ALC}$-axioms in an ontology file using StoreOntology class.
5. ALCLTLGenerator does the following tasks:
   (a) It generates $\mathcal{ALC}$-LTL formulae.

(b) It store the generated $\mathcal{ALC}$-LTL formulae using StoreOntology class.

(c) It loads already generated $\mathcal{ALC}$-LTL formulae using LTLOWLParser class which is explained later.

Utility Classes

Some detailed information about the utility classes is explained as follows:

1. CmdParser does the following tasks:
   (a) It parses input command line arguments,
   (b) It prints command line instructions for the input parameters.
2. LoadOntology does the following tasks:
   (a) It creates a new ontology and store it in provided location,
   (b) It loads an ontology into the internal data structures from a given URL,
   (c) It loads an ontology into the internal data structures from a given File.
3. LTLOWLAxiom An LTLOWLAxiom is an internal data structure representation which helps to store $\mathcal{ALC}$-LTL formulae.
4. StoreOntology does the following tasks:
   (a) It saves an ontology into any of the following format.
      i. OWLXML Format,
      ii. Manchester Format,
      iii. Default Format.
   (b) It stores the provided $\mathcal{ALC}$-axioms in an Ontology,
   (c) It stores the provided $\mathcal{ALC}$-LTL formulae in an XML file,
5. LTLOWLParser does the following tasks:
   (a) It loads $\mathcal{ALC}$-LTL formulae from the given ontology & XML file,
6. PrettyVisitor does the following task:

(a) This class is used to print $\mathcal{ALC}$-axioms and $\mathcal{ALC}$-LTL formulae in their syntactic definition format.

4.2 Program Options

The program presents different options to the end user to perform any of the following tasks:

1. Generate, store and load a set of concepts N_C, set of roles N_R and a set of individuals N_I.
2. Generate, store and load $\mathcal{ALC}$-axioms of the following types:
   (a) $\mathcal{ALC}$-concept assertion axioms.
   (b) $\mathcal{ALC}$-role assertion axioms.
   (c) $\mathcal{ALC}$-negated role assertion axioms.
   (d) GCIs (General Concept Inclusions).
3. Generate, store and load $\mathcal{ALC}$-LTL formulae.

4.3 Results

Generate N_C, N_R, N_I:

N_C, N_R, N_I The following command arguments are used for this operation:

```
-gc [NC_SIZE] [NR_SIZE] [NI_SIZE] -o [FILE_PATH]
```

- NC_SIZE is a positive integer value specifying the randomly generated concept set size.
- NR_SIZE is a positive integer value specifying the randomly generated role set size.
- NI_SIZE is a positive integer value specifying the randomly generated individual set size.

- FILE_PATH specify the location where the output file saved in OWL-XML format.

Example command:

```
-gc 5 5 5 -o /tmp/output.owl
```

The format of output.owl is given in the appendix**??**.

Generate $\mathcal{ALC}$-Axioms:

The following command arguments are used to generate $\mathcal{ALC}$-Axioms:

```
-gf [SIZE] [LENGTH] -i [INPUT_FILE_PATH] -w [OUT_PUT_FILE_PATH]
```

- SIZE specify the number of $\mathcal{ALC}$-axioms should be obtained in the end.
- LENGTH specify the length of each generated $\mathcal{ALC}$-axioms.
- INPUT_FILE_PATH file location which store sets N_C, N_R and N_I in OWL-XML format.
- OUTPUT_FILE_PATH is the file where $\mathcal{ALC}$-axioms are stored in OWL-XML format.

Key Results:

The following shows an example command run and its displayed output result:

```
-gf 5 5 5 -i /tmp/output.owl  -w /tmp/alc.owl

1: r4(a2, a1)
2: a1: ∀r4. ¬(c1 ⊓ ∀r1. (c1 ⊓ c3))
3: (a1: ¬∀r4. (∃r2 . c2 ⊔ ∃r5. c5)
4: (((c1 ⊓ c3) ⊔ ¬∀r5. c5)⊑ ∀r2. c1)
5: ¬r5(a3, a2)
```

The format of alc.owl is given in the appendix [2].

Generate $\mathcal{ALC}$-LTL formulae:

The following command:

```
-gg [SIZE][LENGTH][AX_LENGTH] -i [INPUT_FILE_PATH] -w [OUT_PUT_FILE_PATH]
```

- SIZE specify the number of $\mathcal{ALC}$-LTL should be obtained in the end.
- LENGTH specify the length of each generated $\mathcal{ALC}$-LTL formula.
- AX_LENGTH specify the length of each $\mathcal{ALC}$-axiom occur in $\mathcal{ALC}$-LTL formula.
- INPUT_FILE_PATH file location which contains sets N_C, N_R, N_I in OWL-XML format.
- OUTPUT_FILE_PATH is path where two files having extension *.owl and *.owl.xml stores $\mathcal{ALC}$-LTL formulae.

Key Results:

The following shows an example command run and its displayed output result:

```
-gg 5 3 3 -i /tmp/output.owl -w /tmp/alc_ltl.owl
```

```
1: (X(X((∀r3. (c3 ⊔ c4) ⊑ ∀r2. c5 U ¬∃r4. c1 ⊑ (c4 ⊔ c5))))
2: (((c2 ⊓ (c4 ⊓ c5)) ⊑ ∃r3. c2 U ¬(a5:∀r2. (c1 ⊓ ∀r3. c3)))
3: (X(¬((c2 ⊔ (c2 ⊓ c3)) ⊑ ∀r2. c1 ∨ ¬ r2(a2, a5)))
4: (((r3(a3, a5) ∧ r5(a2, a5)) ∧ X(a1: ¬∃r2. (c1 ⊔ c5)))
5: (¬(r2(a2, a4) ∨ (r3(a3, a5) ∨ a5: ((c1 ⊓ c2) ⊔ (c2 ⊔ c4))))
```

The format of alc_ltl.owl and alc_ltl.owl.xml is given in the appendix [3]. The structure of the document is which store $\mathcal{ALC}$-LTL formulae is as follows:

- Each stored $\mathcal{ALC}$-LTL formula in XML has an opening tag ⟨Formula⟩ and closing tags ⟨/Formula⟩.

- Every temporal opertor occur in an $\mathcal{ALC}$-LTL formula has its own opening and closing tags:
  - Negation has an opening tag and closing tag ⟨NegationOf⟩, ⟨/NegationOf⟩,
  - Conjunction has an opening tag and closing tag ⟨ConjunctionOf⟩, ⟨/ConjunctionOf⟩,
  - Disjunction has an opening tag closing tag ⟨DisjunctionOf⟩, ⟨/DisjunctionOf⟩,
  - Next has an opening tag and closing tag ⟨NextOf⟩, ⟨/NextOf⟩,
  - In Until contains two additional tags ⟨LeftOf⟩ and ⟨RightOf⟩ beside its opening and closing tags ⟨UntilOf⟩ because left and right tags helps to keep track which part of the formula occurs on the left and which one occurs on the right side of the until operator.

Load $\mathcal{ALC}$-LTL formulae:

The following command:

```
-gl -i [INPUT_FILE_PATH]
```

- INPUT_FILE_PATH is the location which contains $\mathcal{ALC}$-LTL formulae in two separate files.

Chapter 5

Conclusions

The emphasis in the report was to explain the implementation details of a program which generate $\mathcal{ALC}$-LTL formulae.

5.1 Future Directions

It is explained in Baader et al. (2008a) that rigid roles and rigid concepts, roles and concepts which do not change over time, play an important role in any temporalized description logic. Therefore incorporating, rigid roles and concepts in $\mathcal{ALC}$-LTL formulae generator is the first task we are looking for in near future. Extending $\mathcal{ALC}$ with a more expressive description logics like $\mathcal{ALCU}$ and $\mathcal{ALNIO}$ could also a possible option where the future work can be directed.

Appendix A

Appendix

1. The following shows a small segment from output.owl which stores N_C, N_R and N_I in OWL-XML format.

```
<?xml version="1.0"?>
<Ontology>
 <Declaration>
  <Class IRI="http://www.semanticweb.org/owlapi:ontology#c1"/>
 </Declaration>
  ...
 <Declaration>
  <ObjectProperty IRI="http://www.semanticweb.org/owlapi:ontology#r1"/>
 </Declaration>
 ...
 <Declaration>
  <NamedIndividual IRI="http://www.semanticweb.org/owlapi:ontology#a1"/>
 </Declaration>
```

```
 ...
</Ontology>
```

2. The following shows a small segment from alc.owl file which stores $\mathcal{ALC}$-axioms in OWL-XML format. An $\mathcal{ALC}$-axiom is given on the top which is followed by a format used to store it.

```
4: (((c1 ⊓ c3) ⊔ ¬∀r5. c5)⊑ ∀r2. c1)


<?xml version="1.0"?>
...
<SubClassOf>
<Annotation>
<AnnotationProperty IRI="SubClassOf"/>
<Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">4</Literal>
</Annotation>
 <ObjectUnionOf>
 <ObjectIntersectionOf>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c1"/>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c3"/>
 </ObjectIntersectionOf>
 <ObjectComplementOf>
   <ObjectAllValuesFrom>
   <ObjectProperty IRI="http://www.semanticweb.org/owlapi:ontology944#r5"/>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c5"/>
   </ObjectAllValuesFrom>
 </ObjectComplementOf>
```

```
 </ObjectUnionOf>
 <ObjectAllValuesFrom>
   <ObjectProperty IRI="http://www.semanticweb.org/owlapi:ontology944#r2"/>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c1"/>
 </ObjectAllValuesFrom>
</SubClassOf>
```

3. The following shows a small segment from alc_ltl.owl.xml file storing $\mathcal{ALC}$-LTL formulae in OWL-XML format. A $\mathcal{ALC}$-LTL formula is given on the top which is followed by a format used to store it.

```
1: (X(X((∀r3. (c3 ⊔ c4) ⊑ ∀r2. c5 U ¬∃r4. c1 ⊑ (c4 ⊔ c5))))
```

```
<?xml version="1.0"?>
...
<Formula>
 <NextOf>
 <NextOf>
 <UntilOf>
 <LeftOf>
  <Axiom>
  <Literal>1</Literal>
 </Axiom>
 </LeftOf>
 <RightOf>
  <Axiom>
  <Literal>2</Literal>
```

```
  </Axiom>
 </RightOf>
 </UntilOf>
 </NextOf>
 </NextOf>
</Formula>
...
```

4. The following shows a small segment from alc_ltl.owl file storing $\mathcal{ALC}$-axioms occur in above $\mathcal{ALC}$-LTL formulae in OWL-XML format.

```
<?xml version="1.0"?>
...
<SubClassOf>
 <Annotation>
 <AnnotationProperty IRI="SubClassOf"/>
  <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">1</Literal>
 </Annotation>
 <ObjectAllValuesFrom>
  <ObjectProperty IRI="http://www.semanticweb.org/owlapi:ontology944#r3"/>
  <ObjectUnionOf>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c3"/>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c4"/>
  </ObjectUnionOf>
 </ObjectAllValuesFrom>
 <ObjectAllValuesFrom>
```

```
  <ObjectProperty IRI="http://www.semanticweb.org/owlapi:ontology944#r2"/>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c5"/>
  </ObjectAllValuesFrom>
</SubClassOf>
...
<SubClassOf>
 <Annotation>
 <AnnotationProperty IRI="SubClassOf"/>
  <Literal datatypeIRI="http://www.w3.org/2001/XMLSchema#string">2</Literal>
 </Annotation>
 <ObjectComplementOf>
  <ObjectSomeValuesFrom>
  <ObjectProperty IRI="http://www.semanticweb.org/owlapi:ontology944#r4"/>
   <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c1"/>
  </ObjectSomeValuesFrom>
 </ObjectComplementOf>
 <ObjectUnionOf>
    <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c4"/>
    <Class IRI="http://www.semanticweb.org/owlapi:ontology944#c5"/>
 </ObjectUnionOf>
</SubClassOf>
...
```

References

Franz Baader, Silvio Ghilardi, and Carsten Lutz. Ltl over description logic axioms. In *Proceedings of the 21st International Workshop on Description Logics (DL2008)*, volume 353 of *CEUR-WS*, 2008a. 1, 5, 6, 7, 30

Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*. Elsevier, 2008b. 3, 4

Sean Bechhofer, Raphael Volz, and Phillip Lord. Cooking the semantic web with the owl api. pages 659–675. Springer, 2003. 8

Michael K. Smith, Chris Welty, and Deborah L. McGuinness. Owl web ontology language guide. Technical report, W3C Recommendation, 2004. 7

C. Lutz, F. Wolter, and M. Zakharyaschev. Temporal description logics: A survey. In *Proceedings of the Fourteenth International Symposium on Temporal Representation and Reasoning*. IEEE Computer Society Press, 2008. 5

Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977. 1, 5