Australian Government
Department of Defence
Defence Science and
Technology Organisation

Survey of Knowledge Representation and Reasoning Systems

Kerry Trentelman

Command, Control, Communications and Intelligence Division

Defence Science and Technology Organisation

DSTO–TR–2324

ABSTRACT

As part of the information fusion task we wish to automatically fuse information derived from the text extraction process with data from a structured knowledge base. This process will involve resolving, aggregating, integrating and abstracting information - *via* the methodologies of Knowledge Representation and Reasoning - into a single comprehensive description of an individual or event. This report surveys the key principles underlying research in the field of Knowledge Representation and Reasoning. It represents an initial step in deciding upon a Knowledge Representation and Reasoning system for our information fusion task.

Published by

DSTO Defence Science and Technology Organisation
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: (08) 8259 5555
Facsimile: (08) 8259 6567

AR No. AR–014–588
July 2009

APPROVED FOR PUBLIC RELEASE

Survey of Knowledge Representation and Reasoning Systems

Executive Summary

As part of the information fusion task we wish to automatically fuse information derived from the text extraction process with data from a structured knowledge base. This process will involve resolving, aggregating, integrating and abstracting information - *via* the methodologies of Knowledge Representation and Reasoning - into a single comprehensive description of an individual or event. This report surveys the key principles underlying research in the field of Knowledge Representation and Reasoning. It represents an initial step in deciding upon a Knowledge Representation and Reasoning system for our information fusion task.

We find that although first-order logic is a highly expressive knowledge representation language, a major drawback of the logic as a Knowledge Representation and Reasoning system for our information fusion task is its undecidability. Moreover, most first-order automated theorem provers are not designed for large knowledge-based applications. Modal logics are gradually receiving more attention by the Artificial Intelligence community, but research in modal logics for knowledge representation still has a long way to go. A production rule (expert) system is viable as a Knowledge Representation and Reasoning system, but these systems are optimally suited for small, specific domains. To build an intelligence expert system we would require expert knowledge in pretty much everything. Frame systems are limited in their expressiveness, and moreover - in regards to knowledge representation - have been superceded by description logics. Semantic networks are excellent for taxonomies, but are not particularly suitable for our information fusion task. On a more positive note, description logics are currently very popular and are actively being researched. They are (in the most part) decidable and their open-world semantics would allow us to represent incomplete information. A further advantage is the availability of Semantic Web technologies. Description logics are still limited however; for our task, we'd need to look at very expressive logics which might lose us decidability.

Author

Kerry Trentelman
Command, Control, Communications and Intelligence Division

Kerry received a PhD in computer science from the Australian National University in 2006. Her thesis topic was the formal verification of Java programs. She joined the Intelligence Analysis Division in 2007 and now works in the area of information fusion. Her research interests include natural language processing, logics for knowledge representation and program specification, and theorem proving and automated reasoning.

Contents

Glossary

ABox A set of assertions describing the properties of various constant symbols in a vocabulary.

Assignment function A function which maps variables to elements in a given model domain. An assignment function can be thought to assign context.

Formula/Concept A type of description which is built in a well-defined way using: elements from a vocabulary; various connectives, punctuation marks and other symbols; sometimes quantifiers; and sometimes a possibly infinite set of variables.

Frame A named list of slots into which values can be placed. Frames can be grouped and organised.

Interpretation An interpretation of a vocabulary element is the semantic value in the model domain assigned to it by the interpretation function. An interpretation of a variable is the value in the model domain assigned to it by the assignment function.

Interpretation function A model's interpretation function maps each symbol in a given vocabulary to a semantic value in the model domain.

Knowledge base A collection of symbolically represented knowledge over which reasoning is performed.

KR&R system A system which represents knowledge symbolically and reasons over the knowledge in an automated way.

Model A situation defined by a pair specifying a non-empty domain and interpretation function. There can be multiple models for a given vocabulary with differing domains and interpretation functions.

Model domain Any set of real or imaginary things which are of interest, *e.g.* individuals, organisations, events, places, or objects.

Production rule An antecedent set of conditions and a consequent set of actions.

Satisfiability Given a model of a particular vocabulary and (if required) an assignment function which maps variables to elements of the model domain, a formula/concept over the same vocabulary is said to be satisfied in the model if a formula-specific configuration of the interpreted formula elements corresponds with the model itself.

Semantic network A directed graph consisting of vertices, which represent objects, individuals, or abstract classes; and edges, which represent semantic relations.

TBox A set of terminological axioms.

Terminological axiom Statements which describe how concepts are related to each other.

Theorem prover A program which determines whether a given formula is valid.

Valid formula Given the set of all possible models for a particular vocabulary, a formula over that same vocabulary is said to be valid if it satisfied in every model of the set given any assignment function.

Vocabulary A unique set of predicate, constant and function symbols.

1 Introduction

Today's intelligence analysts are finding themselves overloaded with information. Valuable information, sometimes implicit, is often buried amongst masses of irrelevant data. As quickly as possible the information must be extracted, cross-checked for accuracy, analysed for significance, and disseminated appropriately. As part of the DSTO's C3ID Intelligence Analysis Discipline, we believe that automating components of this process offers a practical solution to this information overload. We propose an intelligence information processing architecture which includes speech processing, information extraction, data-mining and estimative intelligence components, as well as a fifth information fusion component discussed shortly. An implementation of the architecture, called the Threat Anticipation Intelligence Centre (TAIC), is currently under development.

The TAIC is based on the Unstructured Information Management Architecture. This is a common framework for processing large volumes of unstructured information such as natural language documents, email, audio, images and video [Ferrucci et al. 2006]. Using this framework, analytic applications can be built in order to extract latent meaning, relationships and other relevant facts from unstructured information. We won't go into further detail regarding the TAIC or its architecture here. However, it is relevant to our discussion to describe the information fusion component of the tool. This component intends to automatically fuse - albeit in an intelligent way - information derived from the text extraction process with data from a structured knowledge base. This process will involve resolving, aggregating, integrating and abstracting information - *via* the methodologies of Knowledge Representation and Reasoning - into a single comprehensive description of an individual or event. From such fused information we hope to obtain improved estimation and prediction, data-mining, social network analysis, and semantic search and visualisation.

Knowledge Representation and Reasoning (KR&R) is 'the area of Artificial Intelligence concerned with how knowledge can be represented symbolically and manipulated in an automated way by reasoning programs' [Brachman & Levesque 2004]. Knowledge is represented symbolically by means of a knowledge representation. According to [Davis, Shrobe & Szolovits 1993] these can be best described in terms of the following five notions.

1. A knowledge representation is a surrogate; it is a substitute for the physical object, event or relationship it represents.

2. A knowledge representation is a set of ontological commitments; these denote the terms in which we can think about the world and allow us to focus on the aspects of the world which we believe to be relevant.

3. A knowledge representation is a fragmentary theory of intelligent reasoning; it should provide the set of inferences the representation allows or recommends.

4. A knowledge representation is a medium for efficient computation; it should be processable without being too computationally expensive.

5. A knowledge representation is a medium of human expression; namely, it is a language in which we describe the world.

This report surveys the key principles underlying research in KR&R. We examine various representation languages and reasoning systems with the aim of deciding upon the best approach to our information fusion task. We stress that this report in no way gives a complete overview of the field. There is much ground that we do not cover, some of which is discussed briefly in Section 9, and we only make a shallow pass. As a step towards information fusion however, we feel that an initial survey is worthwhile.

It's worth commenting that much of this report is concerned with logic. Logic is fundamental to many formal knowledge representation and reasoning methodologies because it provides languages for symbolic representation, truth conditions for sentences written in those languages, and rules of inference for reasoning about sentences. The report is outlined as follows. In Section 2 we look at first-order logic, which holds a privileged status amongst the varieties of logic due to its expressivity. In Section 3 we discuss modal logic, which has recently received interest by the Artificial Intelligence community. Section 4 looks at production rule systems. Here knowledge representation is more limited, but reasoning is conducive to procedural control. Section 5 discusses an object-oriented approach to knowledge representation called frames. Section 6 concentrates on description logics, an extension of the frame formalism. Section 7 looks at semantic networks, which have long been used in philosophy, psychology and linguistics, and are now being developed for Artificial Intelligence. Section 8 discusses various KR&R technologies associated with the Semantic Web. Section 9 provides a general discussion. Finally, in Section 10, we draw conclusions and suggest future directions for our information task. We should point out that the 'Implementations' sub-sections describe influential and/or well-known KR&R systems implemented according to the methodologies outlined in the encompassing section. We add these comments merely for reader interest.

2 First-Order Logic

By far, First-Order Logic (FOL) is the most commonly studied and implemented logic. Its invention is usually credited to Gottlob Frege and Charles Sanders Peirce. Frege's concept writing - described in his ground-breaking book *Begriffsschrift*, 1879 - and Peirce's existential graphs - discussed later in Section 7.1 - converged on systems that were semantically identical, but neither were widely accepted. The logic only began to emerge in the 1920s and was eventually adopted by the Artificial Intelligence community for knowledge representation purposes *circa* 1980 [Hayes 1977, Israel 1983]. FOL has only half-a-dozen or so basic symbols, but, depending on the choice of predicates, highly expressive knowledge representations can be constructed using these elements. In this section we define the syntax and semantics of FOL and briefly look at a fragment called propositional logic. After a simple example applied to knowledge representation, we examine the tableau and resolution proof methods, the two main methods of reasoning in FOL. We conclude the section with some additional comments on implementations.

We first define a syntax for the logic. As outlined in [Blackburn & Bos 2005] a vocabulary of a first-order language is comprised of the following.

- A set of unique predicate symbols of arity n such that $n \geq 1$. Often these symbols are denoted using capitalised mixed case, or more generally using P, Q and R with subscripts.

Usually (but not always) the vocabulary also contains:

- A set of unique constant symbols. These symbols are often denoted using uncapitalised mixed case, or more generally using a, b and c with subscripts.
- A set of unique function symbols of arity m such that $m \geq 1$. These are denoted using uncapitalised mixed case, or more generally using f, g and h with subscripts.

Given a particular vocabulary, we build a first-order language over that vocabulary together with the elements listed below.

- An infinite set of variable symbols, often denoted using x, y and z with subscripts.
- The connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction) and $\Rightarrow$ (implication).
- The variable-binding quantifiers $\forall$ (universal) and $\exists$ (existential).
- Left and right parenthesis and the comma.
- Usually an equality symbol.

The two syntactic expressions in FOL are terms and formulae. A term in FOL is defined as follows.

- Constant symbols and variables are terms.
- If f is a function of arity m and $\tau_1, \ldots, \tau_m$ are terms, then $f(\tau_1, \ldots, \tau_m)$ is also a term.
- Nothing else is a term.

If P is a predicate symbol of arity n and $\tau_1, \ldots, \tau_n$ are terms, then $P(\tau_1, \ldots, \tau_n)$ is said to be an atomic formula. If the equality symbol $=$ is considered part of the language and if τ_1 and τ_2 are terms, then $\tau_1 = \tau_2$ is also said to be an atomic formula. A well-formed formula (or WFF, or simply 'a formula') is defined as follows.

- An atomic formula is a WFF.
- If φ and ψ are WFFs, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\varphi \Rightarrow \psi$ are WFFs.
- If φ is a WFF and x is a variable, then $\forall x.\varphi$ and $\exists x.\varphi$ are WFFs.
- Nothing else is a WFF.

A formula can be thought of as a description. There are a few more points worth noting about first-order formulae. First, quantification is permitted only over variables; this is what distinguishes FOL from higher-order logic. Second, a variable occurrence is said to be bound in a formula if it lies within the scope of a quantifier, otherwise is it said to be free. Third, a sentence of FOL is any formula without free variables. Crucially, these sentences are what we use to represent knowledge.

In order to define a semantics for our first-order language, we introduce the notion of a model. A model for a given vocabulary can be thought of as a situation. Formally, a model M for a given vocabulary is a pair (D, F) specifying a non-empty domain D and an interpretation function F. The domain contains the kinds of things we want to talk about, *e.g.* individuals, organisations, events, places, or objects. The interpretation function specifies for each symbol in the vocabulary a semantic value in the domain. Essentially, it provides an interpretation for each symbol in the vocabulary. Each constant symbol a is interpreted as an element of the domain, *i.e.* $F(a) \in D$. For example $F(\mathit{alice})$ is some element of D, which we can specify as an individual that is called Alice. Each predicate symbol P of arity n is interpreted as an n-ary relation over the domain, *i.e.*

$$F(P) \subseteq \underbrace{D \times \ldots \times D}_{n \text{ times}}$$

For example $F(\mathit{Child})$ is some subset of D, which we can specify as the set of children within the domain. Another example is $F(\mathit{RelatedTo})$ which is some subset of $D \times D$, which we can specify as being the set of pairs of individuals in the domain where the first individual in the pair is related to the second. Each function symbol f of arity m is interpreted as an m-ary function over the domain, *i.e.*

$$F(f) \subseteq \underbrace{D \times \ldots \times D}_{m \text{ times}} \rightarrow D$$

For example $F(\mathit{childOf})$ is some function $D \rightarrow D$, which we can specify as being the function which maps an individual to his or her child.

Note that a given vocabulary can be mapped to the same (or even different) domain *via* a different interpretation function, hence it is possible to generate multiple models for a given vocabulary.

Given a particular vocabulary, a model for that vocabulary and a formula over that vocabulary, we are interested in making some kind of evaluation of the formula (description) with respect to the model (situation). So far we have only seen interpretations given to vocabulary elements. In order to interpret the variables of our first-order formulae, we introduce an assignment function μ which maps from the set of variables to the model domain, *i.e.* $\mu(x) \in D$ for variable x and domain D. We then are able to talk about the satisfaction of a formula in the model with respect to a particular assignment function. Before we can formally define this notion of satisfaction, we give two further definitions.

Let $M \equiv (D, F)$ be a model and let μ be an assignment function which maps variables to elements in D. Let τ be a term. We denote the 'interpretation of τ with respect to F and μ' as $I^{\mu}_{F}(\tau)$ and define it as follows.

$$I^{\mu}_{F}(\tau) \equiv \begin{cases} F(\tau) & \text{if } \tau \text{ is a constant or function symbol} \\ \mu(\tau) & \text{if } \tau \text{ is a variable} \end{cases}$$

Now suppose λ is another function which assigns values to variables in M. Let $x, y, z, \ldots$ be the infinite set of variables of our first-order language. Suppose $\lambda(x) \neq \mu(x)$. Suppose however that for each and every variable distinct from x, $\lambda(y) = \mu(y)$ and $\lambda(z) = \mu(z)$, *etc.* Then we say λ is an x-variant of μ. Variant assignments allow us to try out new values for a given variable (x say) while keeping the values assigned to all other variables

the same. We now define the relation $M, \mu \models \varphi$, which can be read 'formula φ is satisfied in model M with respect to assignment μ', as follows.

$$
\begin{array}{lll}
M, \mu \models P(\tau_1, \ldots, \tau_n) & \text{iff} & (I_F^\mu(\tau_1), \ldots, I_F^\mu(\tau_n)) \in F(P) \\
M, \mu \models \neg\varphi & \text{iff} & \text{not } M, \mu \models \varphi \\
M, \mu \models \varphi \wedge \psi & \text{iff} & M, \mu \models \varphi \text{ and } M, \mu \models \psi \\
M, \mu \models \varphi \vee \psi & \text{iff} & M, \mu \models \varphi \text{ or } M, \mu \models \psi \\
M, \mu \models \varphi \Rightarrow \psi & \text{iff} & \text{not } M, \mu \models \varphi \text{ or } M, \mu \models \psi \\
M, \mu \models \forall x.\varphi & \text{iff} & M, \lambda \models \varphi \text{ for all } x\text{-variants } \lambda \text{ of } \mu \\
M, \mu \models \exists x.\varphi & \text{iff} & M, \lambda \models \varphi \text{ for some } x\text{-variant } \lambda \text{ of } \mu \\
M, \mu \models \tau_1 = \tau_2 & \text{iff} & I_F^\mu(\tau_1) = I_F^\mu(\tau_2)
\end{array}
$$

The symbol $\models$ is usually referred to as the satisfaction relation. Note that if term τ is of the form $f(\tau_1, \ldots, \tau_m)$ for a function f of m terms, then $I_F^\mu(\tau)$ is defined to be $F(f)(I_F^\mu(\tau_1), \ldots, I_F^\mu(\tau_m))$.

Since a vocabulary may have many possible models with differing domains and interpretation functions, a formula over that vocabulary may be satisfied in one model and not in another. We write the set of all possible models over a given vocabulary as $\mathcal{M}$. We say a formula is satisfiable if it is satisfied in at least one model of $\mathcal{M}$ (with respect to a given assignment function) and unsatisfiable otherwise. This notion can be extended to finite sets of formulae. A finite set of formulae $\{\varphi_1, \ldots, \varphi_n\}$ is satisfiable if $\varphi_1 \wedge \ldots \wedge \varphi_n$ is satisfiable. Similarly $\{\varphi_1, \ldots, \varphi_n\}$ is unsatisfiable if $\varphi_1 \wedge \ldots \wedge \varphi_n$ is unsatisfiable. Essentially, satisfiable formulae can be thought of as describing conceivable, possible, or realisable situations. Unsatisfiable formulae describe inconceivable, impossible situations. A simple example of an unsatisfiable formula is $\varphi \wedge \neg\varphi$.

We say a formula is valid if it is satisfiable in all models of $\mathcal{M}$ given any variable assignment, and invalid otherwise. The notation $\models \varphi$ is used to indicate that a formula is valid. A simple example of a valid formula is $\varphi \vee \neg\varphi$. In logic, validity is often considered in terms of logical arguments or inferences. We say that an argument with premises $\varphi_1, \ldots, \varphi_n$ and conclusion ψ is valid if and only if whenever all the premises are satisfied in some model, using some variable assignment, then the conclusion is satisfied in that same model using the same variable assignment. We use the notation $\varphi_1, \ldots, \varphi_n \models \psi$ to indicate that the argument with premises $\varphi_1, \ldots, \varphi_n$ and conclusion ψ is valid. We also say that ψ is a logical consequence of $\varphi_1, \ldots, \varphi_n$ or that $\varphi_1, \ldots, \varphi_n$ logically entails ψ. Here the $\models$ symbol refers to a semantic entailment relation rather than a satisfaction relation; the overloading of the symbol is traditional. Importantly, every valid argument $\varphi_1, \ldots, \varphi_n \models \psi$ corresponds to the valid formula $\models \varphi_1 \wedge \ldots \wedge \varphi_n \Rightarrow \psi$. Moreover, two formulae φ and ψ are said to be logically equivalent if and only if both $\varphi \models \psi$ and $\psi \models \varphi$.

We can now define what it means for a sentence to be true (or false) in a model. Recall that sentences do not contain any free variables. Hence their satisfaction will not depend on any given variable assignment. We say a sentence α is true in a model M if and only if for any assignment μ of values to variables in M, we have $M, \mu \models \alpha$. Otherwise α is said to be false in M. If α is true in M, we write $M \models \alpha$. We also use the notation $M \models S$, where S is a set of sentences, meaning that all the sentences of S are true in M.

2.1 Reasoning with FOL

We'll now look at the types of reasoning we can perform with our first-order formulae. As discussed in [Blackburn & Bos 2005] there are three reasoning tasks fundamental to the field of computational semantics: query; consistency checking; and informativity checking. Given a particular vocabulary, a model M for that vocabulary and a first-order formulae φ over that vocabulary, a query task asks whether φ is satisfied in M. As long as the models are finite, the querying task can be straightforwardly handled by a first-order model checker.

Given a particular vocabulary, the set of all possible models $\mathcal{M}$ for that vocabulary and a first-order formula φ over that vocabulary, a consistency check asks whether φ is consistent (meaning that it is satisfied in at least one model of $\mathcal{M}$) or inconsistent (meaning that φ is satisfied in no model of $\mathcal{M}$). We mentioned previously that a formula is said to be satisfiable if it is satisfied in at least one model, hence consistency is usually identified with satisfiability, and inconsistency with unsatisfiability. Consistency checking for first-order formulae is computationally undecidable, meaning that there is no algorithm capable of solving this problem for all input formulae. Not only must a satisfying model be found amongst the vast number of possible models, but that satisfying model must be finite. However, some formulae only have satisfying models which are infinite in size.

Given a particular vocabulary, the set of all possible models $\mathcal{M}$ for that vocabulary and a first-order formulae φ over that vocabulary, an informativity check asks whether φ is informative (meaning that it is not satisfied in at least one model of $\mathcal{M}$) or uninformative (meaning that φ is satisfied in all models of $\mathcal{M}$). Since a formula is invalid if there is at least one model in which it is not satisfied, and is valid if it is satisfied in all models, we usually identify informativity with invalidity and uninformativity with validity. Valid formulae can be seen to be uninformative since they don't tell us anything new about a particular model. Informativity checking for first-order formula is also undecidable.

Derived from their definitions, consistency and informativity are related as follows.

- φ is consistent iff $\neg\varphi$ is informative.
- φ is inconsistent iff $\neg\varphi$ is uninformative.
- φ is informative iff $\neg\varphi$ is consistent.
- φ is uninformative iff $\neg\varphi$ is inconsistent.

For example suppose φ is consistent. This means it is satisfied in at least one model, which is the same as saying that there is at least one model in which $\neg\varphi$ is not satisfied. Hence $\neg\varphi$ is informative. Because of these inter-relations, both (in)consistency and (un)informativity checks can be reformulated in terms of validity. A conclusion ψ is uninformative with respect to premises $\phi_1, \ldots, \phi_n$ if and only if the formula $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \psi$ is valid, whereas ψ is inconsistent with respect to $\phi_1, \ldots, \phi_n$ if and only if $\phi_1 \wedge \ldots \phi_n \Rightarrow \neg\psi$ is valid. For example we can immediately see that a conclusion $Q(a)$ is uninformative with respect to the premises $\forall x.P(x) \Rightarrow Q(x)$ and $P(a)$. To show this formally, we need to check that $\forall x.(P(x) \Rightarrow Q(x)) \wedge P(a) \Rightarrow Q(a)$ is valid, which it obviously is. Similarly, we can see that $Q(a)$ is inconsistent with respect to $\forall x.P(x) \Rightarrow \neg Q(x)$ and $P(a)$. In order to show

this, we'd need to check that $\forall x.(P(x) \Rightarrow \neg Q(x)) \wedge P(a) \Rightarrow \neg Q(a)$ is valid. Again, it is obvious that this is the case. To prove the validity of less obvious first-order formulae, an (undecidable) theorem prover can be used. These programs usually implement tableau or resolution-based proof methods; both methods are discussed briefly in Section 2.1.2.

2.1.1 KR&R aspects

Note that when it comes to knowledge representation and reasoning based on FOL, we are interested in determining the validity of formulae involving sentences. In particular, we are interested in whether a set of sentences S logically entails a sentence α. The set S of sentences used as a basis for calculating entailment can be thought of as a knowledge base (KB). Before looking at the resolution and tableau proof methods used to infer entailment, we'll take a brief look at the internals of a simple knowledge base. We start by defining a vocabulary. This encompasses the kinds of objects (constant symbols) we are interested in reasoning about, the properties the objects are thought to have (unary predicate symbols), the relationships among the objects (n-ary predicate and function symbols). From here, using the knowledge representation language of FOL, we can populate the knowledge base itself.

First we have the constant symbols of our first-order language which, under an interpretation, can be thought of as named individuals, organisations, objects, events or places. For example, suppose we have *alice*, *whiteRabbit*, *madHatter*, *dormouse*, *queenOfHearts*, *kingOfHearts*, *pocketWatch*, *bottleOfPotion*, *pieceOfCake*, *rabbitHole* and *teaParty*. Next we have unary predicate symbols which again, under an interpretation, can be thought of as: the types of things our constant symbols are, *e.g. Child*, *Rabbit*, *Man*, *Rodent*, *Woman*, *TimePiece*, *Potion*, *Cake*, *UndergroundDwelling* and *Event*; and the properties the constant symbols can have, *e.g. NineFeetTall*, *Late*, *Mad*, *Sleepy*, *Violent*, *Sensible* and *Stupid*. Following this, we have the n-ary predicate symbols that under an interpretation express relationships. For example, the binary predicate symbols *FallsDown*, *Drinks*, *Eats*, *Attends*, *BelongsTo*, *FriendOf* and *MarriedTo*, and the 3-ary predicate symbol *GivesTo*. Lastly, we have the function symbols of the domain, *e.g.* the unary symbol *hostOf*. Note that unary function symbols can also be written as binary predicate symbols. Furthermore, all functions are total in FOL, meaning that every element of a function's domain is associated with an element of the codomain. Hence we prefer to specify *hostOf* as a function symbol and *attends* as a binary predicate symbol, since all tea-parties need to be hosted, but not attended.

We can now construct the basic facts of our first-order knowledge base. We can apply types to constant symbols, *e.g.* $Child(alice)$, $Man(madHatter)$, $Rodent(dormouse)$, $Woman(queenOfHearts)$, $TimePiece(pocketWatch)$ and $Event(teaParty)$. Furthermore, we can capture the properties of constant symbols, *e.g.* $Late(whiteRabbit)$, $Sleepy(dormouse)$, $Violent(queenOfHearts)$ and $Stupid(teaParty)$; along with the relationships between constant symbols, *e.g.* $FallsDown(alice, rabbitHole)$, $Drinks(alice, bottleOfPotion)$, $Eats(alice, pieceOfCake)$, $Attends(alice, teaParty)$, $BelongsTo(pocketWatch, whiteRabbit)$, $FriendOf(dormouse, madHatter)$ and $MarriedTo(kingOfHearts, queenOfHearts)$. Moreover, we can specify the functions of our knowledge base by utilising equality in FOL, *e.g.* $hostOf(teaParty) = madHatter$. More complex facts - often referred to as terminological facts - may also be incorporated. Terminological facts, as described in [Brachman &

Levesque 2004], can be classified as follows. Note that we use the abbreviations $\alpha \supset \beta$ for $\neg\alpha \vee \beta$ and $\alpha \equiv \beta$ for $(\alpha \supset \beta) \wedge (\beta \supset \alpha)$, where α and β are both sentences. We also use our natural language interpretation of the predicates *Female*, *Male*, *Queen*, *Ruler*, *etc.*

1. Disjoint facts, where the assertion of one predicate implies the negation of the other, *e.g.* $\forall x.Female(x) \supset \neg Male(x)$.
2. Subtypes, where one predicate is subsumed by another, *e.g.* $\neg\forall x.Queen(x) \supset Ruler(x)$. Here a queen is a type of ruler, but not all rulers are queens.
3. Exhaustive facts, where two or more subtypes completely account for a supertype, *e.g.* $\forall x.Card(x) \supset (Hearts(x) \vee Diamonds(x) \vee Spades(x) \vee Clubs(x))$.
4. Symmetric facts, *e.g.* $\forall x, y.MarriedTo(x, y) \supset MarriedTo(y, x)$.
5. Inverses, where one relationship is the opposite of another, *e.g.* $\forall x, y.ChildOf(x, y) \supset ParentOf(y, x)$.
6. Type restrictions, where the arguments of a predicate must be of a particular type, *e.g.* $\forall x, y.Attends(x.y) \supset Creature(x) \wedge Event(y)$.
7. Full definitions, where compound predicates are fully defined by a logical combination of other predicates, *e.g.* $\forall x.WhiteRabbit(x) \equiv Rabbit(x) \wedge White(x)$.

It's worth commenting that relationships such as *MarriedTo*, *BelongsTo* and *FallsDown* are time-dependent; and even location-dependent in the case of *FallsDown*. It is only during a particular period in time that a king is married to a queen, a rabbit owns a pocketwatch, or a child falls down a rabbit-hole. In order to reflect a more realistic knowledge base we need to incorporate such information. For example, we might have the following

$$MarriedTo(kingOfHearts, queenOfHearts, May1832, July1877)$$
$$FallsDown(alice, rabbitHole, 4thMay9am1965, lat51.751233, long-1.256049)$$

However, for the purposes of this document, we aim to keep things simple and will not include such information here.

We now look at deriving implicit conclusions/entailments from our explicitly represented knowledge base. For example, we can ask 'does the host of the tea-party have any friends?' This can be written in FOL as $\exists x.FriendOf(x, hostOf(teaParty))$? Specifically, we want to see if this sentence is logically entailed by the sentences of the knowledge base. This means we need to determine whether the knowledge base being true in some model implies that the sentence is true in the same model. We start by picking some model M of the knowledge base KB and assume $M \models KB$. It follows that both $hostOf(teaParty) = madHatter$ and $FriendOf(dormouse, madHatter)$ are true in M since these sentences are all in the knowledge base. As a result, the sentence $FriendOf(dormouse, hostOf(teaParty))$ is true in M, and therefore $\exists x.FriendOf(x, hostOf(teaParty))$ is true in M. Hence we can not only determine that the host of the tea-party has a friend, we also know who that friend is; namely the dormouse. Since the same argument applies for any model of the knowledge base, we know that the sentence is indeed entailed by the knowledge base.

Here we have given a rather simplistic example of a knowledge base and an informal proof of entailment. In the next section we'll briefly discuss the tableau and resolution proof methods which can be used to automate the proofs of validity for FOL formulae.

2.1.2 The tableau and resolution proof methods

Given a FOL formula φ, the tableau proof method checks its validity by proving that $\neg\varphi$ is unsatisfiable. Moreover the method checks the validity of an inference with premises $\{\varphi_1, \ldots, \varphi_n\}$ and conclusion ψ by proving the set $\{\varphi_1, \ldots, \varphi_n, \neg\psi\}$ is unsatisfiable. A tree is constructed - called a tableau - such that formulae in nodes of the same branch are conjuncted, whereas different branches are disjuncted. Applicable rules of a tableau calculus are applied in any order and top-down to each node. These rules specify how each logical connective is to be broken down. Complex formulae are eventually broken into atomic formulae - or their negation - until the tree becomes rule-saturated. At this point the tree can no longer be expanded. A branch containing an opposite pair of literals is called closed. By literal we mean an atomic formula, or the negation of an atomic formula. If all branches of the tableau are closed, then we have found a tableau proof for the set of formulae, meaning that the set of formulae is unsatisfiable. We refer the reader to [Blackburn & Bos 2005] for a more detailed discussion on the tableau proof method.

The resolution proof method initially requires a formula of FOL logic to be converted to a variation of the Conjunctive Normal Form (CNF) called set CNF. A formula is said to be in (ordinary) CNF if and only if it is a conjunction of clauses. By clause we mean a disjunction of literals. For example the formula $(p \vee q) \wedge (r \vee \neg p \vee s) \wedge (q \vee \neg s)$ is in CNF. Rewrites are used to transform a formula into CNF. These drive negations and disjunctions deeper into the formula past any quantifiers, eliminate implications and double negations, and 'lift out' conjunctions. Existential quantifiers are skolemized and any universal quantifiers are discarded. Usually clauses are given a list representation, *e.g.* $p \vee q$ is written as $[p, q]$. Furthermore, the connective $\wedge$ is given a list-of-lists representation. Hence our previous example can be written $[[p, q], [r, \neg p, s], [q, \neg s]]$. A list-of-lists is termed a 'clause set'. A clause set is in set CNF if (1) none of its clauses are repeated, and (2) none of its clauses contain repeated literals. A clause set may contain the empty clause $[\,]$ which is always false. Essentially $[\,]$ is logically equivalent to $\bot$, which can be thought of as an atomic formula which is always false in any given model with respect to any given assignment. An important point is that if a formula in CNF (or set CNF) is true, then all of its clauses must be true. Hence if a formula contains an empty clause it cannot be true.

The resolution proof method is based upon the repeated use of what is called the binary resolution rule.

$$\frac{[p_1, \ldots, p_n, r, p_{n+1}, \ldots, p_m] \qquad [q_1, \ldots, q_j, \neg r, q_{j+1}, \ldots, q_k]}{[p_1, \ldots, p_n, p_{n+1}, \ldots, p_m, q_1, \ldots, q_j, q_{j+1}, \ldots, q_k]}$$

Here, given two clauses without repeated literals, $C \equiv [p_1, \ldots, p_n, r, p_{n+1}, \ldots, p_m]$ and $C' \equiv [q_1, \ldots, q_j, \neg r, q_{j+1}, \ldots, q_k]$ say, if C contains a positive literal and C' contains its negation, then we can apply the resolution rule by discarding the pair of literals and merging the remainders to the clause $[p_1, \ldots, p_n, p_{n+1}, \ldots, p_m, q_1, \ldots, q_j, q_{j+1}, \ldots, q_k]$. Note that the merged clause may contain repeated literals; these need to be discarded before

this new clause can be resolved against another by further application of the resolution rule. We can see that the method is satisfaction preserving; if both C and C' are true in some model M, then at least one literal in each clause must be true in M. Since only one of the pair r and $\neg r$ can be true, at least one other literal from either C or C' must be true in M. This literal will feature in the merged clause; hence the merged clause, being a disjunction of literals, will also be true in M.

The general idea behind the resolution proof method is as follows. If we want to show that a formula φ of FOL is valid then we convert $\neg\varphi$ to set CNF and try to generate an empty clause by applying the binary resolution rule. If a clause set contains an empty clause - which is always false in any given model, with respect to any given assignment - then the formula represented by the clause set cannot be satisfied in any model. Therefore, if we generate the empty clause from $\neg\varphi$ *via* the satisfaction-preserving resolution method, then φ must be satisfied in all models, hence φ must be valid. We refer the reader to [Blackburn & Bos 2005] for a more detailed discussion on the resolution proof method.

2.2 Propositional logic

We'll now give a quick sketch of a quantifier-free fragment of FOL called propositional logic. We refer to this logic in later sections. Suppose we have a vocabulary, a model over that vocabulary, and an assignment function μ which assigns values to variables in the model. As pointed out in [Blackburn & Bos 2005], the quantifier-free formulae of propositional logic contain no bound variables; because there are no quantifiers, no variables can lie within their scope. In order to assign truth values to the quantifier-free formulae we therefore consider each free variable x as a constant symbol interpreted by $\mu(x)$. Then every atomic quantifier-free formula effectively becomes an atomic quantifier-free sentence, and hence is either true or false in a model with respect to μ. Moreover, we can specify the semantic value of more complex sentences: conjunctions of formulae are true if and only if all conjuncts are true; disjunctions of formulae are true if and only if at least one disjunct is true; a negated formula is true if and only if the formula itself is not true; and a sentence formed using implication is true if and only if at least one of the negated antecedent or the consequent is true. Truth tables are often constructed for such a purpose. Below we show the truth tables for sentences involving $\wedge$, $\vee$, $\Rightarrow$ and $\neg$.

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$
true	*true*	*true*	*true*	*true*
true	*false*	*false*	*true*	*false*
false	*true*	*false*	*true*	*true*
false	*false*	*false*	*false*	*true*

p	$\neg p$
true	*false*
false	*true*

Figure 1: *Truth tables of propositional logic*

Note that the symbols p, q and r - called propositions - are traditionally used to denote the atomic formulae of propositional logic. This simpler notation is adopted because no free variable needs binding and hence the internal structure of an atomic formula is of no relevance. A propositional calculus is a formal system combining propositional logic with proof rules which allow certain formulae to be established as theorems of the system.

Common proof rules include: *modus ponens*, where from the formulae p and $p \Rightarrow q$ we infer q and *modus tollens*, where from the formulae $\neg q$ and $p \Rightarrow q$ we infer $\neg p$. We refer the reader to [Goré 2003] for a thorough introduction to the logic.

2.3 Implementations

The popular logic programming language Prolog - the name derived from *PROgrammation en LOGique* - is based on the Horn clause subset of FOL. (A Horn clause is a disjunction of literals with at most one positive, *i.e.* non-negated, literal.) The restriction makes Prolog fast enough to be a practical programming language. For more information, see [Colmerauer & Roussel 1993] for a history of the language, or [Blackburn, Bos & Striegnitz 2006] for an introductory online course.

The Knowledge Interchange Format (KIF) is based on FOL and serves as an interchange language between disparate programs [Genesereth & Fikes 1992]. The Suggested Upper Merged Ontology which is the largest formal ontology publicly available today, is written in KIF [Pease 2008]. The interchange language was intended for standardisation by the American National Standards Institute (ANSI), but the effort was abandoned. A framework of FOL-based interchange languages called Common Logic has since been developed and has received approval by the International Organization for Standardization (ISO). More details about Common Logic can be found at [Delugach & Menzel 2007].

3 Modal Logic

Modal logics are logics designed for reasoning about different modes of truth. For example, they allow us to specify what is necessarily true, known to be true, or believed to be true. These modes - often referred to as modalities - include possibility, necessity, knowledge, belief and perception. Among these, the most important are what 'must be' (necessity) and what 'may be' (possibility). As discussed in [Emerson 1990], their interpretation gives rise to different variations of modal logics. For example, if necessity (possibility) is interpreted as necessary (possible) truth, we have alethic modal logic. If necessity (possibility) is interpreted as a moral or normative necessity (possibility), we have deontic logic. If necessity (possibility) is interpreted as referring to that which is known (not known) or to be believed (not believed) to be true, we have epistemic logic. Finally, if necessity (possibility) is interpreted as referring to that which always has been or to which henceforth will always be (possibly) true, we have temporal logic. Note that much of the following section is derived from [Zalta 1995, Cresswell 2001, Halpern 2005].

A modal logic is formed by taking any logic - usually propositional, sometimes FOL or even non-classical logics such as intuitionistic or relevant logic - and augmenting it with logical operators denoting the modalities. In order to keep things simple, here we will focus on an alethic modal logic based on FOL without functions or the equality symbol. Terms are simply constants or variables, and all atomic formulae are of the form $P(\tau_1, \ldots, \tau_n)$, where P is a predicate symbol of arity n and $\tau_1, \ldots, \tau_n$ are terms. The modal operators of our example first-order modal logic are syntactically represented by the diamond $\Diamond$ and box $\Box$ symbols. We define a well-formed formula (or WFF, or simply 'a formula') of the

logic as follows.

- An atomic formula is a WFF.
- If φ and ψ are WFFs, then $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\Diamond\varphi$ and $\Box\varphi$ are WFFs.
- If φ is a WFF and x is a variable, then $\forall x.\varphi$ and $\exists x.\varphi$ are WFFs.
- Nothing else is a WFF.

If φ is a formula, then the first-order modal formula $\Diamond\varphi$ intuitively means 'φ is possibly true', whereas $\Box\varphi$ intuitively means 'φ is necessarily true'. With such a logic we can, for example, represent the following sentences.

- It is possible that the White Rabbit is late.
- It is possible that Alice will grow to be nine feet tall.
- It is not possible that: every queen is violent, the Queen of Hearts is a queen, and the Queen of Hearts is not violent.
- It is necessary that Alice falls down the rabbit-hole or Alice does not fall down the rabbit-hole.

Other modal logics usually have different modal operators. For example in epistemic logic, the basic modal operators are K and C which respectively represent 'it is known that' and 'it is common knowledge that'. A formula such as $\mathsf{K}_{alice}\exists x.Late(x)$ represents Alice knows that someone is late, whereas $\exists x.\mathsf{K}_{alice}Late(x)$ represents Alice knows someone who is late. A formula such as $\mathsf{C}Mad(madHatter)$ represents the common knowledge that the Mad Hatter is mad. By common knowledge we mean that everyone knows the Mad Hatter is mad, everyone knows that everyone knows the Mad Hatter is mad, everyone knows that everyone knows that everyone knows the Mad Hatter is mad, and so on. In temporal logics the basic modal operators are U, X, F and G which respectively represent until, next, eventually and globally. The formulae of linear temporal logic are interpreted over paths/time-lines represented by state transition systems (which we won't discuss further here). A formula such as $Grows(alice)\mathsf{U}NineFeetTall(alice)$ means that Alice is nine feet tall at some current or future position (state) of the path, and that Alice must grow until that position. Moreover at that position, Alice no longer needs to keep growing. The formula $\mathsf{X}Cries(alice)$ means Alice must cry at the next state, whereas $\mathsf{F}Shrinks(alice)$ means Alice has to eventually shrink somewhere along the subsequent path. The formula $\mathsf{G}Child(alice)$ means that Alice has to remain a child along the entire subsequent path.

Attributable to [Kripke 1963], any modal logic can be assigned a possible world semantics. Essentially, a possible world is any world which is considered possible. This includes not only our own, real world, but any imaginary world whose characteristics or history is different. Here we will supply a semantics for our example first-order modal logic. Given a vocabulary of unique constant and predicate symbols, a model M for that vocabulary is a quadruple (W, R, D, V) which consists of: (1) a non-empty set W of possible worlds; (2) an accessibility relation $R \subseteq W \times W$ between worlds, whereby $R(w, w')$ denotes that

world w has access to world w', or that w' is accessible or reachable from w, or that w' is a successor of w; (3) a domain D of the kinds of individuals, places or objects we want to talk about, and which is common to all worlds; and (4) an interpretation function V which assigns a semantic value in D to each symbol in the vocabulary at each world of W. Each constant symbol a is interpreted as a pair consisting of an element of the domain and a world, *i.e.* $V(a) \in D \times W$. For example $V(\mathit{whiteRabbit})$ is some element of $D \times W$, which we can specify as being an individual called the White Rabbit in some particular world. Each predicate symbol P of arity n is interpreted as an n-ary relation, *i.e.*

$$V(P) \subseteq \underbrace{D \times \ldots \times D}_{n \text{ times}} \times W$$

For example $F(\mathit{Late})$ is some subset of $D \times W$, which we can specify as being the set of individuals in the domain who are running late in some particular world.

One assumption we will make in our example modal logic is that our constants are rigid. By this we mean the interpretation of a constant symbol is the same at every world. Hence if we interpret $V(\mathit{whiteRabbit})$ as being an individual called the White Rabbit in some world, then $V(\mathit{whiteRabbit})$ is interpreted as the same individual in any other world. Note however that the interpretation of a predicate symbol at some world may differ from its interpretation at some other world. Hence the notion of 'lateness' might mean being five minutes late in one world, but ten minutes late in another.

As with FOL, we introduce an assignment function μ in order to interpret the variables of our first-order modal logic. This function maps from the set of variables to the model domain, *i.e.* $\mu(x) \in D$ for variable x and domain D. Moreover, for a term τ of our logic, we denote the 'interpretation of τ with respect to V and μ' as $I^{\mu}_{V}(\tau)$ and define it as follows.

$$I^{\mu}_{V}(\tau) \equiv \begin{cases} V(\tau) & \text{if } \tau \text{ is a constant symbol} \\ \mu(\tau) & \text{if } \tau \text{ is a variable} \end{cases}$$

Now given a vocabulary and a model for that vocabulary, every formula over that vocabulary has a truth-value at a world in a model with respect to a particular assignment function. We define the relation $M, w, \mu \models \varphi$, which can be read 'formula φ is true at world w in model M with respect to assignment μ', as follows.

$M, w, \mu \models P(\tau_1, \ldots, \tau_n)$	iff	$(I^{\mu}_{V}(\tau_1), \ldots, I^{\mu}_{V}(\tau_n)) \in V(P)$
$M, w, \mu \models \neg\varphi$	iff	not $M, w, \mu \models \varphi$
$M, w, \mu \models \varphi \wedge \psi$	iff	$M, w, \mu \models \varphi$ and $M, w, \mu \models \psi$
$M, w, \mu \models \varphi \vee \psi$	iff	$M, w, \mu \models \varphi$ or $M, w, \mu \models \psi$
$M, w, \mu \models \varphi \Rightarrow \psi$	iff	not $M, w, \mu \models \varphi$ or $M, w, \mu \models \psi$
$M, w, \mu \models \forall x.\varphi$	iff	$M, w, \lambda \models \varphi$ for all x-variants λ of μ
$M, w, \mu \models \exists x.\varphi$	iff	$M, w, \lambda \models \varphi$ for some x-variant λ of μ
$M, w, \mu \models \Diamond\varphi$	iff	there is a w' such that $R(w, w')$ and $M, w', \mu \models \varphi$
$M, w, \mu \models \Box\varphi$	iff	$M, w', \mu \models \varphi$ for every world w' such that $R(w, w')$

We can now see how the accessibility relation plays a role in the definition of truth. A world w can access a world w' if every formula that is true at w is possibly true at w'. If there are formulae that are true at w', but are not possibly true at w, then that is because w' is not accessible from w, *i.e.* w' represents a state of affairs that is not possible from

the point of view of w. Therefore a formula is necessarily true at a world w if the formula is true at all worlds that are possible from the point of view of w. We should point out that different modal logics result from placing various conditions upon the accessibility relation, *e.g.* reflexivity, symmetry, and transitivity. Furthermore, multi-modal logics can be constructed. Such logics contain multiple accessibility relations. For example the alethic multi-modal logic $\mathbf{K_m}$ has m different accessibility relations $R_1, \ldots, R_m$. Each relation R_i, where $1 \leq i \leq m$, is quantified using the multi-modal operators $\Box_i$ and $\Diamond_i$. We later refer to this logic in Section 6.3. We say a formula φ is true in a model M with respect to assignment μ if $M, w, \mu \models \varphi$ for every world $w \in W$. Recall that we use $\mathcal{M}$ to represent the set of all possible models over a given vocabulary. We say that a formula of our first-order modal logic is valid if it is true in all models of $\mathcal{M}$, given any variable assignment. At this stage it's worth mentioning that reasoning with modal logics is usually performed using refinements of the resolution and tableau proof methods. We won't go into any details here, instead we refer the reader to [Goré 1999, Nivelle, Schmidt & Hustadt 2000] for more information.

Modal logic introduces the notion of a domain of individuals, places or objects which change from state to state (or world to world). In fact, modal logic even allows for things to exist in one world but not another (suppose we had ignored the common domain requirement in our example logic). Although it is not yet researched as actively as FOL or propositional logic, modal logic is gradually receiving more and more attention by the Artificial Intelligence community.

4 Production Rule Systems

No single knowledge representation language is likely to be optimal for all types of systems or all domains. The logics we have discussed thus far (modal, propositional and first-order) are particularly suitable for representing real world models and complex relationships amongst objects and individuals. Production rule systems fail in this regard, however they are ideal for representing procedural knowledge. A production rule system is a reasoning system that uses assertions and rules for knowledge representation. The assertions are maintained in a working memory similar to a constantly evolving database. The rules - called production rules, or simply 'productions' - consist of two parts: an antecedent set of conditions and a consequent set of actions. They are given the following form.

$$IF < conditions > THEN < actions >$$

If a production rule's conditions match the current state of the working memory, then the rule is said to be applicable. The actions are then executed or 'fired', usually resulting in a modified working memory. Intuitively, production rules can be thought of as generative rules which capture the what-you-do-when knowledge. An inference engine is used to (1) determine the set of applicable rules, and (2) prioritise the set when more than one rule is applicable at a time. Note that the majority of this section, including examples, derives from [Brachman & Levesque 2004].

The basic operation of a production system can be summarised in the following three steps.

1. Find which rules are applicable, *i.e.* those rules whose conditions are satisfied by the current working memory.
2. Among the applicable rules found, termed the conflict set, choose which rules should fire.
3. Perform the actions of all the rules chosen to fire and hence modify the working memory.

The cycle repeats until no applicable rules can be found; the system halts at this point. Note that we describe a generic production rule system here. There are many variations, for example, some systems will fire only one rule per cycle.

The working memory is composed of a set of elements, each of which is of the form

$$(type \quad attribute_1 : value_1 \quad \ldots \quad attribute_n : value_n)$$

Here types and attributes are constant symbols, whereas values may be constant symbols or numbers. Examples include $(child \quad age : 10 \quad name : alice)$ and $(timePiece \quad version : pocketWatch \quad belongsTo : whiteRabbit)$. A production rule condition can be either positive or negative. Negative conditions have a minus sign placed in front of them. The body of a condition of a production rule is of the following form.

$$(type \quad attribute_1 : specification_1 \quad \ldots \quad attribute_n : specification_n)$$

Here each specification is one of the following: (1) a variable; (2) a square bracketed evaluable expression; (3) a curly bracketed test; or the conjunction ($\wedge$), disjunction ($\vee$), or negation ($\neg$) of a specification. Note that the precise syntax will vary according to the production rule system and parser being used. For example, the condition $(child \quad age : [n+2] \quad name : x)$ is satisfied if there is a working memory element with type *child*, and whose *age* attribute has the value $n+2$, where n is specified in some other rule condition. If the variable x is already bound, then the element's *name* value needs to match the value of x. Alternatively, if x is not bound, the element binds its *name* value to x. Another example is the negative condition $-(child \quad age : \{\leq 5 \wedge \geq 15\})$ which is satisfied if there is no working memory element with type *child* and *age* value between five and fifteen.

As described in [Brachman & Levesque 2004], a rule of a production rule system is applicable if all the conditions of the rule are satisfied by the working memory. A positive condition is satisfied if there is a matching element in the working memory, whereas a negative condition is satisfied if there is no matching element. An element matches a condition if (1) the types are identical, and (2) for each attribute-value pair, there is a corresponding attribute-value pair in the condition, whereby the value matches the specification according to the assignment of variables.

The actions of a production rule are interpreted procedurally. All actions are to be executed in sequence, and each action is one of the following.

1. An $ADD < pattern >$ action (sometimes referred to as *MAKE*). Here an element specified by $< pattern >$ is added to the working memory.
2. A *REMOVE* i action, where i is an integer. Here the elements matching the i-th condition in the antecedent of the rule are removed.

3. A *MODIFY* i ($<$ *attribute* *specification* $>$) action. Here the element matching the i-th condition in the antecedent of the rule is modified by replacing its current value for *attribute* by *specification*.

Some example sentences which can be formulated using production rules include the following.

- If Alice is ten years old and has a birthday then Alice will be eleven.
- If the Queen of Hearts is angry and violent then she sentences every creature to death.
- If the Queen of Hearts is married to the King of Hearts then the King of Hearts is married to the Queen of Hearts.
- If Alice eats some cake then she grows to be nine feet tall.

The production rule formulation of the latter sentence assumes that some rule has previously added an element of type *eatsCake* to the working memory at the right time.

$$\begin{array}{l} IF\ (child \quad height : x \quad name : alice)\ (eatsCake \quad who : alice) \\ THEN\ \ MODIFY\ 1\ (height \quad 9) \\ \qquad\quad REMOVE\ 2 \end{array}$$

Hence when the *height* value of the working memory element is changed, the *eatsCake* element is removed, so that the rule does not fire again. Note also that there is no distinction between a *MODIFY* action and a *REMOVE* action followed by an *ADD* action. For example, the production rule

$$\begin{array}{l} IF\ (rabbit \quad name : whiteRabbit \quad status : late) \\ THEN\ MODIFY\ 1\ (status \quad onTime) \end{array}$$

has the same effect on the working memory as the following rule.

$$\begin{array}{l} IF\ (rabbit \quad name : whiteRabbit \quad status : late) \\ THEN\ \ REMOVE\ 1 \\ \qquad\quad ADD\ (rabbit \quad name : whiteRabbit \quad status : onTime) \end{array}$$

An inference engine will often employ a particular algorithm called the Rete algorithm. For interest's sake, we provide a short description. The Rete algorithm is an efficient pattern matching algorithm used to determine the conflict set, *i.e.* the set of applicable rules. Designed by Charles Forgy of Carnegie Mellon University, the algorithm - named using the Latin word for network - proved a breakthrough for the implementation of production rule systems. Informally, the idea behind the algorithm is as follows. Because the rules of a production system do not change during its operation, a network of nodes - where each node corresponds to a fragment of a rule condition - can be constructed in advance. While the system is in operation, tokens representing new or changed working memory elements are passed incrementally through the network. Tokens that make it all the way through the network on any given cycle are considered to satisfy all the conditions

of a rule. At each cycle, a new conflict set can then be calculated from the previous one, and any incremental changes can be made to the working memory. Hence only a fraction of the working memory is re-matched against any rule conditions, thereby reducing the computational cost of calculating the conflict set. The inference engine will also employ various conflict resolution strategies in order to determine the most appropriate rule(s) to fire. The strategies might take into account rule priority, the order in which conditions are matched in the working memory, the complexity of each rule, or some other criteria. Often an engine allows users to select between strategies or to chain multiple strategies.

One of the main advantages of production rule systems is their modularity. Each production rule defines an independent piece of knowledge; new rules may be added and old rules may be deleted (usually) independently of other rules. Another advantage is that rules may be easily understood by non-experts. Disadvantages arise from the inefficiency of large systems with unorganised rules.

4.1 Implementations

Mycin was an early (1970s) production rule system designed to diagnose infectious blood diseases and recommend antibiotics. (The name derives from the suffix '-mycin' of many antibiotics.) Mycin is deemed an 'expert' system; these are production rule systems which contain subject-specific knowledge of human experts. Developed at Stanford University by Edward Shortliffe and others, Mycin would query the user *via* a series of yes or no questions, and output a list of possible culprit bacteria ranked from high to low based on the probability of each diagnosis [Shortliffe 1981]. Although Mycin was never used in practice because of legal and ethical reasons, it proved highly influential to the development of subsequent expert systems.

Jess is a rule engine for the Java platform. Designed by Ernest Friedman-Hill at Sandia National Laboratories, Jess provides rule-based programming suitable for automating an expert system, and is often referred to as a 'expert system shell' (hence the name). Lightweight and fast, it uses an enhanced version of the Rete algorithm to process rules, and features working memory queries. More information can be found at [Friedman-Hill 2007].

5 The Frame Formalism

The original idea behind the frame formalism is this: when a person encounters a stereotypical situation or object, they respond to it by using a frame. A frame can be thought of a remembered framework which can be adapted to fit a given situation by changing the aspects of the frame as necessary [Minsky 1975]. Although originally intended for scene-analysis systems, the applicability of frames has a wider scope, in particular to the field of KR&R. Much of the following description of the frame formalism derives from [Brachman & Levesque 2004].

Frames can be thought of as named lists of slots into which values can be placed. The values that fill the slots are called fillers. There are two types of frames: individual and generic frames. Individual frames represent single objects, whereas generic frames

represent categories or classes of objects. We give the syntax of an individual frame as follows.

$$\begin{array}{l}(\mathit{frameName} \\ \quad <: \mathit{slotName1} \quad \mathit{filler1} > \\ \quad <: \mathit{slotName2} \quad \mathit{filler2} > \ldots)\end{array}$$

Here the frame and slot names are constant symbols and the fillers are either constant symbols or numbers. The notation we use here gives the names of individual frames in uncapitalised mixed case, generic frames in capitalised mixed case, and slot names in capitalised mixed case prefixed with a colon. An instantiated example of an individual frame is given below.

$$\begin{array}{l}(\mathit{queenOfHearts} \\ \quad <: \mathit{INSTANCE-OF} \quad \mathit{Queen} > \\ \quad <: \mathit{Likes} \quad \mathit{tart} > \\ \quad <: \mathit{Orders} \quad \mathit{chopOffHead} > \ldots)\end{array}$$

Individual frames have a distinguished slot called $: \mathit{INSTANCE-OF}$. This slot's filler is the name of the generic frame indicating the category of the object being represented. The individual frame can be thought of as being an instance of the generic frame. Hence following our example, the Queen of Hearts is an instance of a queen. Generic frames have a syntax similar to individual frames. We give an example below.

$$\begin{array}{l}(\mathit{Queen} \\ \quad <: \mathit{IS-A} \quad \mathit{RoyalMonarch} > \\ \quad <: \mathit{Sex} \quad \mathit{female} > \\ \quad <: \mathit{Orders} \quad \mathit{Execution} > \ldots)\end{array}$$

Here the slot fillers can be either generic frames or individual frames. Instead of an $: \mathit{INSTANCE-OF}$ slot, a generic frame has a distinguished slot called $: \mathit{IS-A}$. This slot's filler is the name of a more general generic frame. The generic frame can be thought of as being a specialisation of the more general frame. Following our example, a queen is a specialisation of a monarch.

The frame formalism allows us to structure our knowledge. We can think of frames as being knowledge objects, which we group and organise depending on what that knowledge is about. Some example sentences which we can represent using frames include the following.

- All hatters are mad.
- Alice is a child.
- The White Rabbit owns a pocketwatch.
- Children are people.
- People eat cake.
- Tea is served at a tea-party.

Slots of generic frames can also have attached procedures - often called demons - as fillers. The procedures attached to a given slot are usually prefixed by $IF-ADDED$, $IF-NEEDED$ or $IF-REMOVED$. The syntax of such a slot is as follows.

$$<: SlotName \quad [prefix \quad \{method\}] >$$

Here *method* represents some attached procedure, whereas *prefix* represents one of the following: $IF-ADDED$, $IF-NEEDED$ or $IF-REMOVED$. The procedure attached to a given slot prefixed by $IF-ADDED$ is executed in response to a value of the slot being added. The procedure attached to a given slot prefixed by $IF-NEEDED$ is executed in response to a value of the slot being needed, and the procedure attached to a given slot prefixed by $IF-REMOVED$ is executed in response to a value of the slot being emptied.

5.1 Reasoning with frames

Much of the reasoning that is done by a frame system involves creating individual instances of generic frames, filling some of their slots with values and inferring other values. Often a generic frame is used to fill in the values not listed explicitly in an instance. In other words, instance frames inherit default information from their generic versions. Hence *queenOfHearts* inherits a : *Sex* slot from *Queen*. If we had provided no filler for the : *Orders* slot of *queenOfHearts*, then we would know, by inheritance, that we need an instance of *Execution*. Inheritance is defeasible, meaning that an inherited value can always be overridden by a filler. Therefore a filler in a generic frame can be overridden in its instances and specialisations. For example, if we have the following generic frame, we are saying that instances of *RoyalMonarch* have a certain : *Plays* and : *Orders* value by default.

$$\begin{array}{l}(RoyalMonarch \\ \quad <: IS-A \quad Monarch > \\ \quad <: Plays \quad croquet > \\ \quad <: Orders \quad pardon > \ldots)\end{array}$$

However we might also have the following two frames.

$$\begin{array}{l}(Queen \\ \quad <: IS-A \quad RoyalMonarch > \\ \quad <: Sex \quad female > \\ \quad <: Orders \quad Execution > \ldots)\end{array}$$

$$\begin{array}{l}(queenOfHearts \\ \quad <: INSTANCE-OF \quad Queen > \ldots)\end{array}$$

Here *queenOfHearts* inherits an ability to play croquet from *RoyalMonarch*, but also inherits the job of ordering executions from *Queen*, overriding the default granting of pardons by *RoyalMonarch*. Note also that individual frames are allowed to be instances of - and generic frames are allowed to be specialisations of - more than one generic frame. For example

$$\begin{array}{l}(Queen \\ \quad <: IS-A \quad RoyalMonarch > \\ \quad <: IS-A \quad Ruler > \ldots)\end{array}$$

As outlined in [Brachman & Levesque 2004], the basic reasoning performed by a frame system can be summarised by the following algorithmic loop.

1. Either a user or an external system declares that an object or situation exists, thereby instantiating an individual frame which is an instance of some generic frame.
2. Any slot fillers that are not provided explicitly but can be inherited by the new frame instance are inherited.
3. For each slot with a filler, any $IF-ADDED$ procedure that can be inherited is run. This possibly causes new slots to be filled, or new frames to be instantiated, whereby we repeat Step 1.

If the user or external system requires the filler of a slot, then the algorithm proceeds as follows.

1. If there is a filler stored in the slot, then that value is returned.
2. Otherwise, if there is no slot filler that can be inherited, any $IF-NEEDED$ procedure that can be inherited is run. This calculates a filler for the slot, but potentially causes other slots to be filled, or new frames to be instantiated.

If neither of these algorithms produce a filler for a given slot, then the value of the slot is considered unknown. It's worth mentioning that other demons such as $IF-NEW$, $RANGE$ and $HELP$ can be implemented. An $IF-NEW$ procedure can be triggered whenever a new frame is created. $RANGE$ can be run whenever a new value is added, and the procedure will return true as long as the value satisfies the range constraint specified for the slot. $HELP$ can be run whenever the demon is triggered and returns false.

It is argued in [Hayes 1979] that most of the frame formalism is simply a new syntax for a fragment of FOL. Here the word 'most' represents the purely declarative information of the frame, and the fragment of FOL contains only the existential quantifier and conjunction, *i.e.* no universal quantifier, no negation, disjunction nor implication. The frame formalism also lacks the ability to express relationships between properties of the same frame or different frames (although attached procedures could enforce these relationships if required). Even though frames do not add further expressiveness, *cf.* FOL, there are two ways in which frame-based systems have an advantage over systems using FOL. First, they allow us to express knowledge in an object-oriented way. Second, by using only a fragment of FOL, frame-based systems offer more efficient means for decidable reasoning. These two advantages are incorporated into description logics, which formalise the declarative part of frame-based systems. These logics emerged from the development of the frame-based system KL-ONE and are discussed in the next section.

6 Description Logic

Description Logics (DL) are a family of knowledge representation languages called description languages. They are designed as an extension of semantic networks and frames,

equipping these methodologies with a formal, logic-based semantics. Description logics are heavily employed by the Semantic Web community; in particular, they provide the basis for the Web Ontology Language discussed later in Section 8.3. In this section we define the syntax and semantics of DL, and delve into the language hierarchy. We discuss reasoning in DL which is usually performed *via* a tableau-based algorithm, and conclude the section with some comments on the implementation KL-ONE. Much of the following section derives from [Baader & Nutt 2003, Baader 2003, Nardi & Brachman 2003].

There are three types of nonlogical symbols in DL which form the vocabulary.

- Constant symbols, which denote named 'individuals', meaning people, organisations, objects, events or places. These are usually written in uppercase, *e.g. ALICE*, *DUCHESS*, *CHESHIRECAT*, *BABY* and *PIECEOFCAKE*.
- Atomic concepts, which denote the types of things the constant symbols are, and the properties the symbols have. Atomic concepts are usually written in capitalised mixed case, *e.g. Person*, *Child*, *Woman*, *Mother*, *Female*, *Cat*, *Vanishing* and *Cake*.
- Atomic roles, which denote binary relationships. These are usually written in uncapitalised mixed case, *e.g. eats*, *ownsCat* and *hasChild*.

Note that all nonlogical symbols within a given vocabulary should be unique.

Note also that DL has two special atomic concepts $\top$ (top) and $\bot$ (bottom). The additional logical symbols incorporated along with the vocabulary dictate the type of description language. For example, the minimal description language of practical interest is the Attributive Language $\mathcal{AL}$. All description languages are built from this base language. Along with a vocabulary, $\mathcal{AL}$ features the following symbols.

- The constructors $\sqcap$ (intersection) and $\neg$ (complement).
- The universal (value) restrictor $\forall$ and the existential (value) restrictor $\exists$.
- Left and right parenthesis, and the comma.
- The constructors $\sqsubseteq$ (subsumption) and $\doteq$ (equality).

Note that the constructors $\sqcap$ and $\neg$ correspond to the FOL connectives $\wedge$ and $\neg$ of conjunction and negation. Likewise, the restrictors $\forall$ and $\exists$ correspond to the FOL universal and existential quantifiers. Moreover, the constructor $\doteq$ corresponds to the equality symbol $=$ of FOL. We simply present the standard DL terminology and syntax. If we use R to range over roles, C to range over concepts, and A to range over atomic concepts, then the concepts of $\mathcal{AL}$ are defined as follows.

- Every atomic concept is a concept.
- If C_1 and C_2 are concepts, then $C_1 \sqcap C_2$ is a concept.
- If A is an atomic concept, then $\neg A$ is a concept.
- If R is a role and C is a concept, then $\forall R.C$ is a concept.

- If R is a role and $\top$ is the top atomic concept, then $\exists R.\top$ is a concept.
- Nothing else is a concept.

Note that we'll give some examples shortly. All roles in $\mathcal{AL}$ are atomic since the language does not provide for role constructors. Furthermore, in $\mathcal{AL}$, complements may only be applied to atomic concepts and we have only limited existential quantification, *i.e.* the top atomic concept may only feature in the scope of the existential restrictor.

A concept denotes the set of all individuals satisfying the properties specified in that concept. For example, the concept *Person* represents the set of people, whereas $\mathit{Child} \sqcap \mathit{Female}$ represents the set of female children. A concept such as $\neg \mathit{Cat}$ represents all the individuals which are not cats. Note that the intersection and complement of concepts are often referred to as concept conjunction and negation respectively. Regarding the restrictors, if the concept C represents some class of individuals, then the value restriction $\forall R.C$ represents those individuals who are R-related only to the individuals of that class. For example, $\forall \mathit{eats}.\mathit{Cake}$ represents those individuals who only eat cake, whereas $\forall \mathit{eats}.\bot$ represents those individuals who never eat. The value restriction $\exists R.\top$ represents all individuals that are R-related to at least one other individual. For example, $\exists \mathit{ownsCat}.\top$ represents all individuals who own at least one cat.

We will not limit ourselves to a discussion of the $\mathcal{AL}$ language here. We can extend $\mathcal{AL}$ with a variety of logical symbols forming new description languages. These are traditionally named by concatenating with $\mathcal{AL}$ letters corresponding to the new constructors. For example, $\mathcal{ALC}$ is the extension of $\mathcal{AL}$ with complex complements; within this language if C is a (not necessarily atomic) concept, then $\neg C$ is considered a concept also. $\mathcal{ALUEN}$ - which we will later provide a semantics for - is the extension of $\mathcal{AL}$ with a union constructor, full existential quantification and number restriction. Within a $\mathcal{U}$ extension the union constructor $\sqcup$ is added to the language. If C_1 and C_2 are concepts, then $C_1 \sqcup C_2$ is also considered a concept. To give an example, $\mathit{Cat} \sqcup \mathit{Child}$ represents the set of all cats and children. Within an $\mathcal{E}$ extension full existential quantification is added to the language. If R is a role and C is a concept, then $\exists R.C$ is considered a concept. To give an example, $\exists \mathit{ownsCat}.\mathit{Vanishing}$ represents those individuals who own at least one vanishing cat. Within an $\mathcal{N}$ extension, the symbols $\leq$ and $\geq$, and the positive integers are added to the language. If R is a role and n is a positive integer, then both $\leq nR$ and $\geq nR$ are considered concepts. To give an example, $(\geq 3\mathit{hasChild}) \sqcap (\leq 2\mathit{ownsCat})$ represents those individuals who have at least three children and who own at most two cats. Note that it is possible to write union and full existential quantification in terms of complex negation, since the semantics enforce equivalence between $C_1 \sqcup C_2$ and $\neg(\neg C_1 \sqcap \neg C_2)$, and $\exists R.C$ and $\neg(\forall R.\neg C)$. Hence $\mathcal{ALUE}$ is indistinguishable from $\mathcal{ALC}$ and we write $\mathcal{ALCN}$ instead of $\mathcal{ALUEN}$, $\mathcal{ALCN}$ being the extension of $\mathcal{AL}$ with complex complements and number restriction.

In DL, there are five types of syntactic expressions: concepts and roles (which we've already seen) constants and assertions (which we'll look at shortly) and terminological axioms. Terminological axioms make statements about how concepts or roles are related to each other; they are formulated as follows.

- If C_1 and C_2 are concepts, then $C_1 \sqsubseteq C_2$ is a terminological axiom.
- If C_1 and C_2 are concepts, then $C_1 \doteq C_2$ is a terminological axiom.

The former class of axioms are usually referred to as inclusions, the latter equalities. (Terminological axioms themselves are often called sentences.) An inclusion $C_1 \sqsubseteq C_2$ has the meaning that concept C_1 is subsumed by concept C_2, specifically, all individuals that satisfy C_1 also satisfy C_2. For example, the sentence $\mathit{Child} \sqsubseteq \mathit{Person}$ says that any child is also a person. An equality $C_1 \doteq C_2$ has the meaning that the two concepts C_1 and C_2 are equivalent, namely that all individuals that satisfy C_1 are precisely those that satisfy C_2. Often equalities are referred to as definitions. Definitions introduce symbolic names for complex descriptions, which can be used as abbreviations in other descriptions. For example $\mathit{Mother} \doteq \mathit{Woman} \sqcap \exists \mathit{hasChild}.\mathit{Person}$ associates the description on the right-hand side of the equality with the name *Mother*, and the equality $\mathit{Parent} \doteq \mathit{Mother} \sqcup \mathit{Father}$ utilises this definition.

As long as no symbolic names are defined twice in a given set of acyclic definitions, the set of definitions and other terminological axioms becomes a TBox. (An acyclic definition of a name is not defined in terms of itself, nor in terms of other concepts that indirectly refer to itself.) Some example sentences which we can represent in the TBox include the following.

- A hatter is a person who makes only hats.
- Queens are royal monarchs who sometimes like to chop off everyone's head.
- Hares are not rabbits and *vice versa*.
- A girl is a female child.
- Tarts are one aspect of a queen's diet.
- A tart-thief is a person who has stolen at least one tart.
- A dormouse is a sleepy creature.

Informally, the TBox contains a terminology, *i.e.* the vocabulary used throughout the knowledge base. A knowledge base based on DL is comprised of two components, the TBox and the ABox. The ABox contains assertions about individuals formulated in terms of the vocabulary. In the ABox, properties are asserted about unique individuals (constant symbols). Using a and b to range over individuals, R to range over roles and C to range over concepts, then we can make ABox assertions of the form $C(a)$ and $R(a, b)$. The concept assertion $C(a)$ has the meaning that a is described by the concept of C, whereas the role assertion $R(a, b)$ has the meaning that b is a 'filler' of the role R for a. For example $\mathit{Cat}(\mathit{CHESHIRECAT})$ means the Cheshire Cat is actually a cat; and $\mathit{ownsCat}(\mathit{DUCHESS}, \mathit{CHESHIRECAT})$ means the Duchess owns a cat, namely the Cheshire Cat. The ABox is a finite set of such assertions.

We are now ready to define a semantics for $\mathcal{ALUEN}$. As for FOL, a model M for a given vocabulary is the pair (D, F) specifying a non-empty domain D and an interpretation function F. The domain contains the kinds of things we want to talk about, *e.g.* people, organisations, objects, events or places. The interpretation function specifies for each symbol in the vocabulary a semantic value in the domain. Each constant symbol a is interpreted as an element of the domain, *i.e.* $F(a) \in D$. For example $F(\mathit{ALICE})$ is some

element of D, which we can specify as an individual called Alice. Each atomic concept A is interpreted as a set of domain elements, *i.e.* $F(A) \subseteq D$. For example $F(\mathit{Child})$ is some subset of D, which we can specify as the set of all children within the domain. Each role R is interpreted as a binary relation over the domain, *i.e.* $F(R) \subseteq D \times D$. For example $F(\mathit{hasChild})$ is some subset of $D \times D$, which we can specify as being the set of pairs of individuals in the domain where the second individual in the pair is a child of the first.

The set $F(C)$ denoting the interpretation of a concept C is called an extension. We can extend the definition of F to all concepts as follows. Note that we use standard set notation: $\emptyset$ (empty set), $\cap$(intersection), $\cup$ (union) and $\setminus$ (complement).

- For the distinguished top concept $\top$, $F(\top) \equiv D$.
- For the distinguished bottom concept $\bot$, $F(\bot) \equiv \emptyset$.
- $F(C_1 \sqcap C_2) \equiv F(C_1) \cap F(C_2)$.
- $F(C_1 \sqcup C_2) \equiv F(C_1) \cup F(C_2)$.
- $F(\neg C) \equiv D \setminus F(C)$.
- $F(\forall R.C) \equiv \{x \in D \mid \text{for any } y, \text{ if } (x, y) \in F(R) \text{ then } y \in F(C)\}$.
- $F(\exists R.C) \equiv \{x \in D \mid \text{there is at least one } y \text{ such that } (x, y) \in F(R) \text{ and } y \in F(C)\}$.
- $F(\leq nR) \equiv \{x \in D \mid \text{the cardinality of } \{y \mid (x, y) \in F(R)\} \leq n\}$.
- $F(\geq nR) \equiv \{x \in D \mid \text{the cardinality of } \{y \mid (x, y) \in F(R)\} \geq n\}$.

We now define the relation $M \models \alpha$, which can be read 'α is satisfied in M', as follows.

$$\begin{array}{lll} M \models C & \text{iff} & F(C) \not\equiv \emptyset \\ M \models C_1 \sqsubseteq C_2 & \text{iff} & F(C_1) \subseteq F(C_2) \\ M \models C_1 \doteq C_2 & \text{iff} & F(C_1) \equiv F(C_2) \\ M \models C(a) & \text{iff} & F(a) \in F(C) \\ M \models R(a, b) & \text{iff} & (F(a), F(b)) \in F(R) \end{array}$$

For $\mathcal{T}$ a set of terminological axioms, *i.e.* a TBox, we use the notation $M \models \mathcal{T}$ to mean that all axioms in $\mathcal{T}$ are satisfied in M. We say M is a model of $\mathcal{T}$. Likewise for an ABox: for $\mathcal{A}$ a set of assertions, we write $M \models \mathcal{A}$ to mean that all assertions in $\mathcal{A}$ are satisfied in M, and we say M is a model of $\mathcal{A}$.

6.1 Expressive description logics

We'll now take a quick look at the more expressive description languages. These languages feature complex roles such as $R_1 \sqcap R_2$ (intersection), $R_1 \sqcup R_2$ (union), $\neg R$ (complement), $R_1 \circ R_2$ (composition), R^+ (transitive closure) and R^- (inverse). Some examples include $\mathit{hasSon} \sqcup \mathit{hasDaughter}$ which can be used to define *hasChild*, $\mathit{hasHusband} \circ \mathit{hasBrother}$ which can be used to define *hasBrotherInLaw*, the transitive closure of *hasChild* which is

the role *hasDescendant*, and the inverse of *hasChild* which is the role *hasParent*. Additional concepts occurring in expressive description languages include $\leq nR.C$ and $\geq nR.C$ (qualified number restriction) and $\{a_1, \ldots, a_n\}$ (set concepts, for $n \geq 1$). Examples include $\geq 2 ownsCat.Vanishing$ which expresses the set of individuals who own at least two vanishing cats and $\{ALICE, MADHATTER, DORMOUSE, MARCHHARE\}$ which can be used to define the concept of attendees at some tea-party. As before, the description languages are named according to their expressiveness. In order to avoid description languages with long names, $\mathcal{S}$ was introduced as an abbreviation of the $\mathcal{ALC}$ language with transitive roles. The extension $\mathcal{H}$ allows role hierarchies, *i.e.* role inclusions $R_1 \sqsubseteq R_2$, and the extension $\mathcal{R}$ incorporates role intersection. $\mathcal{I}$ allows for inverse roles, whereas $\mathcal{O}$ allows for set concepts. The extension $\mathcal{Q}$ incorporates qualified number restriction. The extension (D) allows the integration of an arbitrary concrete domain within a description language. The domains may be, for example, the set of non-negative integers, spatial regions or even time intervals. An example of an expressive description language is $\mathcal{SHOIN}(D)$ which provides the basis for the web ontology language OWL DL discussed in Section 8.3.

6.2 Reasoning with DL

Typical reasoning tasks for a terminology (TBox) are to determine (1) whether a particular concept is satisfiable, *i.e.* non-contradictory; and (2) whether one concept is more general than another, *i.e.* whether the first description subsumes the second. Given a terminology and the set of all possible models $\mathcal{M}$ for that terminology (whereby the models have differing domains and interpretation functions), we say that a concept C is satisfiable if it is satisfied in at least one model of $\mathcal{M}$. Concept C is unsatisfiable if it is satisfied in no model of $\mathcal{M}$. A concept C_1 is said to be subsumed by a concept C_2 if $C_1 \sqsubseteq C_2$ is satisfied in every model of $\mathcal{M}$. Two concepts C_1 and C_2 are said to be equivalent if $C_1 \doteq C_2$ is satisfied in every model of $\mathcal{M}$. From their definitions, we can reformulate satisfiability and equivalence in terms of subsumption.

- C is unsatisfiable iff C is subsumed by $\bot$.
- C_1 and C_2 are equivalent iff $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$.

Moreover, as long as full concept negation and conjunction are allowed in the description language, subsumption and equivalence can be reformulated in terms of satisfiability.

- C_1 is subsumed by C_2 iff $C_1 \sqcap \neg C_2$ is unsatisfiable.
- C_1 and C_2 are equivalent iff both $C_1 \sqcap \neg C_2$ and $\neg C_1 \sqcap C_2$ are unsatisfiable.

For simple description languages with little expressitivity, *i.e.* which do not allow negation at all, the subsumption of concepts can be computed using structural subsumption algorithms. Such algorithms compare the syntactic structure of concept descriptions. We refer the reader to [Baader & Nutt 2003] for an in-depth discussion. For more expressive description languages, *i.e.* which allow concept negation and conjunction, tableau-based algorithms are used to determine the satisfiability of descriptions. We will shortly give an example of a tableau-based algorithm for the $\mathcal{ALC}$ language.

Typical reasoning tasks for the ABox are to determine (1) whether a particular assertion is consistent, *i.e.* whether it is satisfied in some model; and (2) whether one particular individual is described by some concept. We say that an ABox $\mathcal{A}$ is consistent with respect to a TBox $\mathcal{T}$ if there is at least one model which is both a model for $\mathcal{T}$ and a model for $\mathcal{A}$. We simply say $\mathcal{A}$ is consistent if it is consistent with respect to the empty TBox. An example is the set of assertions $\{\mathit{Mother}(\mathit{DUCHESS}), \mathit{Father}(\mathit{DUCHESS})\}$ which is consistent with respect to the empty TBox, but inconsistent with respect to a TBox defining the disjoint concepts *Mother* and *Father*. Given an ABox $\mathcal{A}$ and the set of all possible models $\mathcal{M}$ for $\mathcal{A}$, we say an assertion α is entailed by $\mathcal{A}$ if α is satisfied in every model of $\mathcal{M}$. Hence $\mathit{Mother}(\mathit{DUCHESS})$ is entailed by an ABox if the Duchess is a mother in every model for that ABox. Derived from their definitions, consistency and entailment are related as follows.

- α is entailed by $\mathcal{A}$ iff $\mathcal{A} \cup \{\neg\alpha\}$ is inconsistent.

Importantly, concept satisfiability in the TBox can be reduced to consistency in the ABox, because for every concept C and arbitrarily chosen individual name a, we have the following.

- C is satisfiable iff $\{C(a)\}$ is consistent.

Hence a concept is satisfiable if it denotes a non-empty set in the model. Provided full concept negation and conjunction are allowed within a given language, all reasoning tasks can thus be reduced to the ABox consistency problem. Note that it is possible to extend tableau-based satisfiability algorithms to algorithms which decide ABox consistency; we refer the reader to [Baader & Nutt 2003] for details.

Informally, the idea behind a tableau-based satisfiability algorithm for $\mathcal{ALC}$ runs as follows. Given a concept C_0, the algorithm tries to construct a model $M \equiv (D, F)$ such that C_0 is satisfied in M, *i.e.* $F(C_0) \not\equiv \emptyset$. The algorithm arbitrarily chooses an individual a where $a \in F(C_0)$. Starting with the ABox $\mathcal{A}_0 \equiv \{C_0(a)\}$, the algorithm then applies consistency-preserving tableau rules until the ABox is complete, *i.e.* until no more rules can be applied. If the resulting ABox does not contain a contradiction, termed a clash, then $\mathcal{A}_0$ is consistent (and thus C_0 is satisfiable) and inconsistent (unsatisfiable) otherwise. Note that a clash in an ABox $\mathcal{A}$ is such that either $\{\bot(a)\} \subseteq \mathcal{A}$, or $\{C(a), \neg C(a)\} \subseteq \mathcal{A}$ for some individual name a and concept C.

Following the example presented in [Franconi 2002], suppose we want to determine whether $C_0 \equiv (\forall \mathit{ownsCat}.\mathit{Vanishing}) \sqcap (\exists \mathit{ownsCat}.\neg \mathit{Vanishing})$ is unsatisfiable. We choose an arbitrary individual a and start with ABox

$$\mathcal{A}_0 \equiv \{((\forall \mathit{ownsCat}.\mathit{Vanishing}) \sqcap (\exists \mathit{ownsCat}.\neg \mathit{Vanishing}))(x)\}$$

We then apply the following tableau rule.

$\sqcap$-rule
Condition: $\mathcal{A}_i$ contains $(C_1 \sqcap C_2)(x)$, but not both $C_1(x)$ and $C_2(x)$.
Action: $\mathcal{A}_{i+1} \equiv \mathcal{A}_i \cup \{C_1(x), C_2(x)\}$.

After applying this rule, we have

$$\mathcal{A}_1 \equiv \mathcal{A}_0 \cup \{(\forall ownsCat.Vanishing)(x), (\exists ownsCat.\neg Vanishing)(x)\}$$

We next apply the tableau rule below.

$\exists$-rule
Condition: $\mathcal{A}_i$ contains $(\exists R.C)(x)$, but there is no z s.t. $C(z), R(x,z)$ are in $\mathcal{A}_i$.
Action: $\mathcal{A}_{i+1} \equiv \mathcal{A}_i \cup \{C(y), R(x,y)\}$ where y does not occur in $\mathcal{A}_i$.

Thus we have

$$\mathcal{A}_2 \equiv \mathcal{A}_1 \cup \{(\forall ownsCat.Vanishing)(x), \neg Vanishing(y), ownsCat(x,y)\}$$

The universal rule is as follows.

$\forall$-rule
Condition: $\mathcal{A}_i$ contains $(\forall R.C)(x)$ and $R(x,y)$, but it does not contain $C(y)$.
Action: $\mathcal{A}_{i+1} \equiv \mathcal{A}_i \cup \{C(y)\}$.

After application of this rule we have

$$\mathcal{A}_3 \equiv \mathcal{A}_2 \cup \{ Vanishing(y), \neg Vanishing(y)\}$$

This complete ABox $\mathcal{A}_3$ contains a clash. This means that our original ABox $\mathcal{A}_0$ is inconsistent and the concept $(\forall ownsCat.Vanishing) \sqcap (\exists ownsCat.\neg Vanishing)$ is unsatisfiable.

6.2.1 Closed *vs.* open-world semantics

It's worth comparing the open-world semantics of DL against the closed-world semantics of a FOL, frame or production rule system. Within a closed-world system nothing is considered true unless it is stated to be true in the knowledge base. For example, an assertion such as $ownsCat(DUCHESS, CHESHIRECAT)$ in a closed-world system expresses that the Duchess has only one cat, the Cheshire Cat. Here the information is considered complete. Within an open-world system it is assumed that not all knowledge is represented in the knowledge base. Hence the assertion $ownsCat(DUCHESS, CHESHIRECAT)$ in an open-world/DL system only expresses that the Cheshire Cat is a cat owned by the Duchess. Here the information is considered incomplete and indicates a lack of knowledge. Note that we can assert the Duchess owns only one cat by $(\leq 1 ownsCat)(DUCHESS)$.

To further illustrate, we follow the example presented in [Baader & Nutt 2003]. Suppose we have the following ABox.

$FriendOf(MADHATTER, DORMOUSE)$	$FriendOf(MARCHHARE, DORMOUSE)$
$FriendOf(MARCHHARE, MADHATTER)$	$FriendOf(ALICE, MARCHHARE)$
$Mad(MADHATTER)$	$\neg Mad(ALICE)$

Suppose we want to find out if the following assertion is true: the Dormouse has a friend who is mad, and that friend in turn has a friend who is not mad. This can be expressed as the question

$$(\exists FriendOf.(Mad \sqcap \exists FriendOf.\neg Mad))(DORMOUSE)?$$

First of all, we know - in all models of our ABox - that the Mad Hatter is a friend of the Dormouse and the Mad Hatter is mad. We know the March Hare is a friend of the Mad Hatter, but we don't know whether the March Hare is *not* mad. Second, the March Hare is a friend of the Dormouse, but again we don't know if the March Hare *is* mad. Using such reasoning in a closed-world system, we cannot claim that the assertion about the Dormouse is true. However, in a open-world system, the models of the ABox can be divided into two classes: one in which the March Hare is mad, and another in which he isn't. In the former type of model, the March Hare, who is mad, is a friend of the Dormouse, and the March Hare has a friend, Alice, who isn't mad. In the latter type of model, the Mad Hatter, who is mad, is a friend of the Dormouse, and the Mad Hatter has a friend, the March Hare, who isn't mad. Therefore in all models of the ABox, the Dormouse has a friend who is mad, who in turn has a friend who is not mad.

Compared with closed-world systems, open-world systems are more easily extendable and reusable. Open-world systems allow us to represent incomplete knowledge; an advantage since we deal with such knowledge every day. Compared with open-world systems, closed-world systems are more easily constrained, hence reasoning over these systems is less complex.

6.3 DL and other KR languages

It's worth commenting on the relation between DL and other logical formalisms. We can see that DLs are essentially fragments of FOL. The constant symbols of DL correspond to the constant symbols of FOL, and since an interpretation F assigns to every atomic concept and role a unary and binary relation over D, respectively, we can view atomic concepts and roles, in FOL terms, as unary and binary predicates. As shown in [Borgida 1996], the language $\mathcal{ALC}$ for example, corresponds to a fragment of FOL called $\mathcal{L}^2$. This fragment is obtained by restricting the FOL syntax to formulae containing two variables and only unary and binary predicates. The translation of $\mathcal{ALC}$ concepts into $\mathcal{L}^2$ formulae relies on two mappings υ_x and υ_y in two free variables x and y, respectively. Each atomic concept A is viewed in $\mathcal{L}^2$ as a unary predicate symbol, whereas each role R is viewed as a binary predicate symbol. Since $\mathcal{ALC}$ is equivalent to the language $\mathcal{ALUE}$ ($\mathcal{AL}$ plus the union constructor and full existential quantification), the translation can be given as follows.

$$
\begin{array}{rcl}
\upsilon_x(A) & = & A(x) \\
\upsilon_x(\neg A) & = & \neg A(x) \\
\upsilon_x(C_1 \sqcap C_2) & = & \upsilon_x(C_1) \wedge \upsilon_x(C_2) \\
\upsilon_x(C_1 \sqcup C_2) & = & \upsilon_x(C_1) \vee \upsilon_x(C_2) \\
\upsilon_x(\exists R.C) & = & \exists y.R(x,y) \wedge \upsilon_y(C) \\
\upsilon_x(\forall R.C) & = & \forall y.R(x,y) \Rightarrow \upsilon_y(C)
\end{array}
\qquad
\begin{array}{rcl}
\upsilon_y(A) & = & A(y) \\
\upsilon_y(\neg A) & = & \neg A(y) \\
\upsilon_y(C_1 \sqcap C_2) & = & \upsilon_y(C_1) \wedge \upsilon_y(C_2) \\
\upsilon_y(C_1 \sqcup C_2) & = & \upsilon_y(C_1) \vee \upsilon_y(C_2) \\
\upsilon_y(\exists R.C) & = & \exists x.R(y,x) \wedge \upsilon_x(C) \\
\upsilon_y(\forall R.C) & = & \forall x.R(y,x) \Rightarrow \upsilon_x(C)
\end{array}
$$

Also worth pointing out is that $\mathcal{ALC}$ directly corresponds to the alethic multi-modal logic $\mathbf{K_m}$ based on propositional logic [Sattler, Calvanese & Molitor 2003]. Recall from Section 3 that this logic has m different accessibility relations $R_1, \ldots, R_m$. Each relation R_i, where $1 \leq i \leq m$, is quantified using the multi-modal operators $\Box_i$ and $\Diamond_i$. The translation of $\mathcal{ALC}$ concepts using role names $R_1, \ldots, R_m$ into $\mathbf{K_m}$ formulae is given by the mapping γ. Each atomic concept A is viewed in $\mathbf{K_m}$ as a unary predicate symbol,

whereas each role R_i is viewed as an accessibility relation.

$$\begin{aligned}
&\gamma(A) = A \\
&\gamma(\neg A) = \neg\gamma(A) \\
&\gamma(C_1 \sqcap C_2) = \gamma(C_1) \wedge \gamma(C_2) \\
&\gamma(C_1 \sqcup C_2) = \gamma(C_1) \vee \gamma(C_2) \\
&\gamma(\forall R_i.C) = \Box_i\gamma(C) \\
&\gamma(\exists R_i.C) = \Diamond_i\gamma(C)
\end{aligned}$$

The $\mathbf{K_m}$ models are viewed as $\mathcal{ALC}$ models and *vice versa*. Informally, an $\mathcal{ALC}$ concept C is satisfied in a model M if its translation $\gamma(C)$ is true in the $\mathbf{K_m}$ model corresponding to M.

Another aspect is that we can add rules to a DL-based knowledge base. Varying somewhat from the *IF* < *conditions* > *THEN* < *actions* > production rules described in Section 4, DL rules are usually written in the form $C_1 \Rightarrow C_2$ for concepts C_1 and C_2. Informally, such a rule has the meaning 'if an individual can be described by a concept C_1, then derive that it can also be described by a concept C_2'. As described in [Baader & Nutt 2003], if we start with an initial knowledge base KB^0 consisting of an initial ABox $\mathcal{A}^0$ and TBox $\mathcal{T}^0$, then we can construct a series of knowledge bases $KB^0, KB^1, \ldots, KB^n$ where KB^{i+1} is obtained from KB^i by adding a new assertion $C_2(a)$ for individual a whenever $KB^i \models C_1(a)$, and as long as KB^i does not contain $C_2(a)$. Note that $KB^i \models C_1(a)$ has the meaning that $C_1(a)$ is satisfied in all models of both $\mathcal{T}^i$ and $\mathcal{A}^i$. This process will eventually halt, since the initial knowledge base contains a finite number of individuals and there are only a finite number of rules. The final knowledge base KB^n will have the same TBox as KB^0, but its ABox will contain additional assertions introduced by the rules. Conventionally, KB^n is termed the procedural extension of KB^0.

6.4 Implementations

Originating from Ronald J. Brachman's PhD thesis [Brachman 1977], the frame-based knowledge representation system KL-ONE acted as the precursor to the field of description logics. This system introduced many key DL notions such as: concepts and roles, and how they are interrelated; number and value restriction; and inference *via* subsumption. KL-ONE also provided the groundwork for the distinction between ABox and TBox, as well as influencing a host of other significant notions in subsequent DL generations. For a more up-to-date discussion on KL-ONE see [Brachman & Schmolze 1985, Nardi & Brachman 2003].

7 Semantic Networks

A semantic network is a directed graph consisting of vertices, which represent objects, individuals, or abstract classes; and edges, which represent semantic relations. Semantic networks have long been used in philosophy, psychology and linguistics - an early example is the tree of Porphyry from the third century A.D. - however implementations for artificial intelligence and machine translation were developed in the early 1960s. Logical expressions can be represented using particular variations of semantic networks called existential graphs and conceptual graphs.

7.1 Existential graphs

Charles Sanders Peirce's existential graphs can be thought of as a graphical or diagrammatic system of logic. Peirce originally proposed three systems of graphs in 1896: Alpha, Beta and Gamma. He continued to develop them until his death in 1914. Alpha and Beta correspond to propositional and first-order logic with equality, respectively. The Gamma system was intended to represent modal logic, but Peirce's work was left incomplete; see [Øhrstrøm 1997] for further discussion. In this section we focus on the Alpha system of existential graphs.

First we define a syntax. As outlined in [Barwise & Hammer 1994] a graph is one of the following.

- A blank page. (Peirce termed the page a sheet of assertion.)
- Symbols denoting propositions, which can be written, *i.e.* asserted, anywhere on the page.
- Subgraphs enclosed by a closed curve called a cut.

Cuts may be empty or nested, but cannot intersect. A semantics is provided as follows.

- The blank page denotes truth.
- Propositions and graphs can be either true or false.
- Juxtaposed propositions are considered conjuncted.
- To surround subgraphs with a cut is equivalent to logical negation; hence an empty cut denotes falsity.
- All subgraphs within a given cut are implicitly conjuncted.

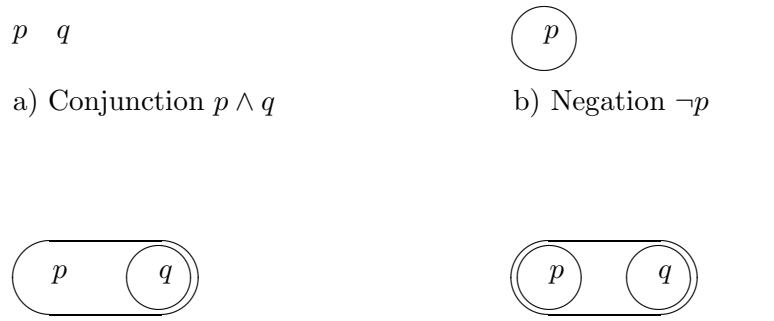
Note that the number of cuts enclosing a subgraph is said to be the depth of that subgraph; a blank page has depth zero. Below we present aspects of the notation with their corresponding representation in propositional logic. It's worth mentioning at this point the logical equivalence of $p \Rightarrow q$ and $\neg(p \wedge \neg q)$, $p \vee q$ and $\neg(\neg p \wedge \neg q)$. We refer the reader to Section 2 for a definition of logical equivalence.

As discussed in [Dau 2003] Peirce provided a propositional calculus for existential graphs by developing five inference rules and one axiom for each system (Alpha, Beta and Gamma). The axiom - the empty graph is assumed to be true - is the same for each system. The rules are both sound and complete for the Alpha and Beta systems. For the alpha system, the five inference rules (Erasure, Insertion, Iteration, Deiteration and Double Cut) are as follows.

Erasure Any graph at an even depth (including depth zero) may be erased.

Insertion Any graph may be inserted into an odd-numbered depth.

Iteration Any copy of a graph may be added at the same depth or deeper.



a) Conjunction $p \wedge q$ b) Negation $\neg p$

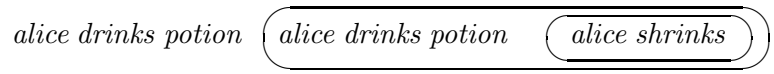
c) Implication $p \Rightarrow q$ d) Disjunction $p \vee q$

Figure 2: *Alpha Existential Graphs*

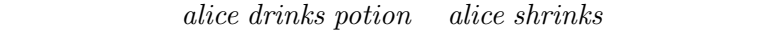
Deiteration Any graph whose occurrence is the result of iteration may be erased.

Double Negation Any double cut (corresponding to a double negation) may be inserted around or removed from any graph.

Below we give a simple example, taken from [Dau 2003], of a proof of *modus ponens* using existential graphs; recall from Section 2.2 that this argument allows us to infer q from the propositional formulae p and $p \Rightarrow q$. Suppose we have the following graph.

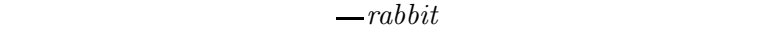


Essentially, this graph has the meaning 'Alice drinks some potion, and if Alice drinks some potion, then she shrinks'. Since the inner instance of *alice drinks potion* can be thought of as a copy of the outer instance, we can apply the Deiteration rule and erase it from the graph. We then have a double cut surrounding *alice shrinks* which can be removed by applying the Double Negation rule, leaving us with

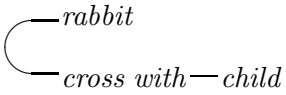


From here we can apply the Erasure rule, erasing the proposition *alice drinks potion* such that *alice shrinks* only remains.

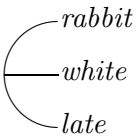
The Beta system builds upon the Alpha system. Predicates of arity n may be used and a new symbol, the line of identity is introduced. A line before a predicate denotes existence. For example,



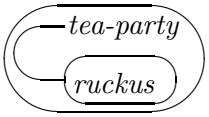
represents 'there is a rabbit'. A line of identity, either curved or straight, connecting two or more predicates asserts that the predicates share a common variable. Such lines can be thought of as showing anaphoric references. For example,



has the meaning 'there is a rabbit who is cross with a child'. There are two lines of identity here, each of which corresponds to an existential quantifier. The graph's first-order logic representation is $\exists x, y.Rabbit(x) \wedge Child(y) \wedge CrossWith(x, y)$. Moreover,



can be thought of as having three lines of identity representing 'there is a rabbit', 'there is something that is white' and 'there is something that is late'. The syntactic junction however asserts that all three predicates share the same variable; in other words 'there is a white rabbit who is late'. The graph's first-order logic representation is $\exists x, y, z.Rabbit(x) \wedge White(y) \wedge Late(z) \wedge x = y \wedge y = z$. After simplification this becomes $\exists x.Rabbit(x) \wedge White(x) \wedge Late(x)$. Another example, following [Sowa 2003], is the following graph which has the meaning 'if there is a tea-party, then there is a ruckus'.



We can give this a first-order representation, namely $\neg(\exists x.TeaParty(x) \wedge \neg Ruckus(x)))$. This representation has the meaning that 'it is false that there is a tea-party and not a ruckus'. Using contraposition between $\neg(\exists x.P(x))$ and $\forall x.\neg P(x)$, we can rewrite this as $\forall x.(\neg(TeaParty(x) \wedge \neg Ruckus(x)))$ and, using a logical equivalence mentioned previously, this becomes $\forall x.(TeaParty(x) \Rightarrow Ruckus(x))$.

7.2 Conceptual graphs

Based on existential graphs and developed by John F. Sowa, conceptual graphs describe a formal language which was originally designed to simplify the mapping to and from natural languages [Sowa 1984]. The graphs correspond to first-order logic; a translation is given in [Sowa 2001]. Following the example given in [Sowa 2000] the sentence 'every rabbit owns two pocket-watches' can be represented by the following conceptual graph.

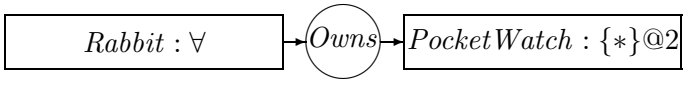


Figure 3: *Conceptual Graph*

Note that we can equivalently write the conceptual graph in a linear fashion, *i.e.*

$$[Rabbit : \forall] \rightarrow (Owns) \rightarrow [PocketWatch : \{*\}@2]$$

In a conceptual graph, boxes (or square-bracketed material) represent objects, individuals or abstract classes; and circles (parenthesised material) represent relations. The boxes are often referred to as concepts. Both concepts and relations are given a natural language semantics. Separated by the colon ':', the left-hand side of each box is a type field which contains a type label, *e.g. Rabbit* or *PocketWatch*. On the right is a referent field, which may (1) be empty, (2) contain a name, (3) contain a universal quantifier such as $\forall$, or (4) contain a plural specification such as $\{*\}@2$. For example the concept $[Rabbit : \forall]$ represents the phrase 'every rabbit', whereas $[PocketWatch : \{*\}@2]$ represents 'two pocket-watches'. Here $\{*\}$ is used to indicate a (plural) set of things, whereas the symbol @ indicates the number of individuals in a referent. An arrow of a conceptual graph is said to belong to a relation and be attached to a concept. Moreover, the conceptual relations of a graph show how referents of the concepts are related. Hence our example graph reads 'every rabbit owns two pocket-watches'. Conceptual graphs may be blank; such graphs contain no concepts, relations or arcs and therefore say nothing about anything. Graphs may also consist of a single concept with no relations or arcs.

Another example is the following.

$$[Girl] \leftarrow (Agnt) \leftarrow [FallsDown] \rightarrow (Dest) \rightarrow [RabbitHole]$$

Here the concept $[Girl]$ has no referent and simply represents 'a girl' or 'there is a girl'. In general, an arrow pointing towards a circle can be read *has a*, whereas an arrow pointing away from a circle can be read *which is a*. Hence our example can be read as two sentences: '*FallsDown* has an agent which is a girl' and '*FallsDown* has a destination which is a rabbit hole'. More simply, the entire graph can be read 'there is a girl who falls down a rabbit hole'. Note that all relations have (1) a type, *i.e.* the name given to the relation, (2) a valence which asserts the number of arrows associated with the relation, and (3) a signature which lists the concept types involved in the relation. For example, consider the graph for 'Alice is between a rock and a hard place' where the relation representing 'between' has type *Betw*, valence 3 and signature $< Thing, Thing, Person >$. (Admittedly this example is contrived; we assume that only a person can be between two things.)

$$\begin{aligned} [Person : Alice] \leftarrow (Betw) \quad & \leftarrow 1 - [Thing : Rock] \\ & \leftarrow 2 - [Thing : Place] \rightarrow (Attr) \rightarrow [Hard] \end{aligned}$$

Here the subgraph $[Thing : Place] \rightarrow (Attr) \rightarrow [Hard]$ can be read '*Place* which has an attribute *Hard*'. Both *Thing* and *Person* are the types of the concepts that are attached to each of the arrows that belong to the relation (*Betw*). For a relation with valence n, the convention is that the first $n-1$ arrows point towards the relation, while the last arrow points away. Furthermore, if $n > 2$ then the first $n-1$ arrows are numbered. The first and last arrows are determined according to the relation's signature. The i-th arrow - where $1 \leq i \leq n-1$ - must be attached to a concept whose concept type is listed in the i-th position of the signature. In our case, *Betw* is specified such that the first two arrows belonging to the relation must attach to concepts which are either of type *Thing* or are of a subtype of *Thing*. Moreover, the third arrow must attach to a concept of type *Person*.

Conceptual graphs may be assigned a context. These are concepts with a nested graph that describes the referent. The example below - adapted from [Sowa 2001] - shows two contexts; one of type *Proposition*, the other of type *Situation*. The overall graph expresses

that 'Alice believes that the Queen of Hearts wants to execute everyone'.

$$[Person : Alice] \leftarrow (Expr) \leftarrow [Believes] \rightarrow (Thme) \rightarrow$$
$$[Proposition : [Person : QueenOfHearts * x] \leftarrow (Expr) \leftarrow [Wants] \rightarrow (Thme) \rightarrow$$
$$[Situation : [?x] \leftarrow (Agnt) \leftarrow [Executes] \rightarrow (Thme) - [Everyone]]]$$

Here the concept $[Person : QueenOfHearts]$ is marked with a coreference-defining label $*x$ which indicates that the subsequent occurrence - marked with the bound label $?x$ - refers to the same person. The graph tells us that Alice is the 'experiencer' $(Expr)$ of the concept $[Believes]$, which is linked by the theme relation $(Thme)$ to a proposition that Alice believes. The proposition contains another conceptual graph, which says that the Queen of Hearts is the experiencer of $[Wants]$, which has as it theme the situation which - according to Alice - the Queen of Hearts wants to create. That situation is described by another nested graph, which says that the Queen of Hearts - represented by the concept $[?x]$ - executes everyone. Note that a coreference between concepts within a conceptual graph corresponds to an existential graph's line of identity.

Reasoning with conceptual graphs can be performed using adaptations of Peirce's Alpha or Beta rules of inference [Kremer & Lukose 1996]. Here we will focus on the adaptations of Alpha rules for conceptual graphs. Note however that the adapted Beta rules deal with the erasure, insertion, iteration, *etc.* of coreferences within conceptual graphs.

In order to explain how the Alpha rules are reformulated in terms of conceptual graphs, we first give some definitions following [Schärfe 2007]. To negate a conceptual graph, the convention is to place a context around it - such as *Situation* or *Proposition* - and then place a negation symbol $\neg$ before the context. Such a context is called negative, *e.g.*

$$\neg[Situation : [QueenOfHearts] \leftarrow (Agnt) \leftarrow [Executes] \rightarrow (Thme) \rightarrow [Everyone]]$$

This graph has the meaning 'it is not the case that the Queen of Hearts executes everyone'. In order to conjunct conceptual graphs we juxtapose the graphs within the same context, *e.g.*

$$[Proposition :$$
$$[QueenOfHearts] \rightarrow (Attr) \rightarrow [Violent]$$
$$[QueenOfHearts] \rightarrow (Attr) \rightarrow [Likes] \rightarrow (Thme) \rightarrow [Croquet]]$$

Such a graph tells us that 'the Queen of Hearts is violent and likes croquet'. In order to disjunct conceptual graphs g and h we use the logical equivalence between $p \vee q$ and $\neg(\neg p \wedge \neg q)$. We place a context around g and negate it, a context around h and negate that, and then place both (negative) contexts inside an even larger context which we also negate, *e.g.* we represent 'the Queen of Hearts plays crouqet or shouts' in conceptual graph form as

$$\neg[Situation :$$
$$\neg[Situation : [QueenOfHearts] \leftarrow (Agnt) \leftarrow [Plays] \rightarrow (Thme) \rightarrow [Croquet]]$$
$$\neg[Situation : [QueenOfHearts] \leftarrow (Agnt) \leftarrow [Shouts]]]$$

Similarly, in order to show implication $g \Rightarrow h$ between two conceptual graphs g and h we use the logical equivalence between $p \Rightarrow q$ and $\neg(p \wedge \neg q)$. We place a context around h and

negate it, and then place both this negative context along with g inside a larger context which we also negate.

A conceptual graph is said to be evenly enclosed if we encounter an even number - including zero - of negative contexts moving outwards from the innermost to the outermost graph, *e.g.*

$$\neg[Proposition : \neg[Proposition : [Hatters : \forall] \rightarrow (Attr) \rightarrow [Mad]]]$$

Here the innermost graph $[Hatters : \forall] \rightarrow (Attr) \rightarrow [Mad]$ is evenly enclosed. Note that we may encounter non-negative contexts, but the term 'even' only applies to negative contexts. (In our case, the graph describing 'all hatters are mad' is also said to be doubly negated, since it has one negated context nested directly inside another.) Furthermore, a graph is said to be oddly enclosed if we encounter an odd number of negative contexts moving from the innermost to outermost graph. A final definition is that of domination. A context u is said to dominate another context v - negated or otherwise - if v is nested in u.

We can now state Peirce's five Alpha inference rules reformulated for conceptual graphs.

Erasure Any evenly enclosed graph may be erased.

Insertion Any graph may be inserted in any oddly enclosed context.

Iteration Any copy of a graph may be added to the same context u in which the original graph occurs, or to any context dominated by u.

Deiteration Any graph whose occurrence is the result of iteration may be erased.

Double Negation Any double negation may be inserted around or removed from any graph in any context.

As an example we'll again show a proof of *modus ponens*; recall that this allows us to infer q from $p \Rightarrow q$ and p. Consider the following graph. Note that again we use the logical equivalence between $p \Rightarrow q$ and $\neg(p \wedge \neg q)$.

$$\begin{array}{l}[Proposition : \\ \quad \neg[Proposition : [KnaveOfHearts] \leftarrow (Agnt) \leftarrow [Steals] \rightarrow (Thme) \rightarrow [Tarts] \\ \quad\quad \neg[Proposition : [KnaveOfHearts] \leftarrow (IsA) \leftarrow [Thief]]] \\ \quad [KnaveOfHearts] \leftarrow (Agnt) \leftarrow [Steals] \rightarrow (Thme) \rightarrow [Tarts]]\end{array}$$

This has the meaning 'if the Knave of Hearts steals some tarts, then he is a thief, and the Knave of Hearts steals some tarts'. Since the inner $[KnaveOfHearts] \leftarrow (Agnt) \leftarrow [Steals] \rightarrow (Thme) \rightarrow [Tarts]$ is a copy of the outer graph, we can apply the Deiteration rule and erase the copy. We can then apply the Double negation rule, removing the double negation surrounding the graph $[KnaveOfHearts] \leftarrow (IsA) \leftarrow [Thief]$. Hence we are left with

$$\begin{array}{l}[Proposition : \\ \quad [KnaveOfHearts] \leftarrow (IsA) \leftarrow [Thief] \\ \quad [KnaveOfHearts] \leftarrow (Agnt) \leftarrow [Steals] \rightarrow (Thme) \rightarrow [Tarts]]\end{array}$$

Since $[KnaveOfHearts] \leftarrow (Agnt) \leftarrow [Steals] \rightarrow (Thme) \rightarrow [Tarts]$ is evenly enclosed, we can erase it by applying the Erasure rule, leaving us with

$$[Proposition : [KnaveOfHearts] \leftarrow (IsA) \leftarrow [Thief]]$$

This final graph has the meaning 'the Knave of Hearts is a thief'.

The Conceptual Graph Interchange Format is a representation for conceptual graphs which has been designed for transmitting graphs across networks. We refer the reader to [Sowa 2001] for a description. The format has been granted ISO standardisation as part of the Common Logic framework referred to in Section 2.3.

7.3 Implementations

The semantic network WordNet is a lexical database of the English language [Al-Halimi et al. 1998]. Developed at the Cognitive Science Laboratory at Princeton University, WordNet contains nouns, adjectives and adverbs grouped into sets of cognitive synonyms called synsets, each of which express a distinct concept. Synsets are linked by semantic and lexical relations and the resulting network can be navigated with a browser. Building on the Princeton WordNet effort, the Global Wordnet Association coordinates the development of wordnets of other languages [Vossen & Fellbaum 2007].

Hermann Helbig's MultiNets - derived from 'Multilayered extended Networks' - is a knowledge representation system based on semantic networks [Helbig 2005]. Natural language expressions are represented by conceptual structures (networks). Concepts are represented by nodes and relations between concepts are represented as arcs between nodes. Every node is classified according to a predefined ontology and is embedded in a multi-dimensional space of attributes and their values. Arcs are labelled from a set of about 150 standardised relations. MultiNet comes with a set of tools including a semantic interpreter which automatically translates German natural language expressions into MultiNet networks, and a browser and parser for managing and graphically manipulating the structures.

8 The Semantic Web

The goal of the Semantic Web initiative is to 'create a universal medium for the exchange of data where data can be shared processed by automated tools as well as by people' [Herman 2001]. Instigated by the World Wide Web Consortium, the initiative aims to extend the World Wide Web using various standards and technology which support richer search, data integration and automation of tasks. As described in [Lukasiewicz 2007] the initiative aims to (1) add machine-readable meaning to web pages using ontologies, thereby assigning a precise definition to shared terms, (2) make use of automated reasoning within a knowledge representation context, and (3) apply cooperative agent technology for processing the information of the web. Hence a major aspect of the Semantic Web vision is the application of KR&R research to the World Wide Web.

Some of the significant technologies of the Semantic Web thus-far realised are: the extensible markup language XML; the resource description framework RDF; the XML and

RDF schema languages; a variety of serialisation notations such as RDF/XML, Turtle, N3 and N-triples; the RDF query language SPARQL; and the Web ontology language OWL. Crucially, these technologies allow us to represent machine-processable semantics of information on the Web. Over the next few sections, we'll give a brief introduction to each Semantic Web technology.

8.1 XML and XML Schema

XML - derived from 'eXtensible Markup Language' - is a general purpose markup language which supports the sharing of structured data across different information systems [Armstrong 2001]. Note that a markup language combines text with extra information, in the form of tags, specifying the structure and layout of the text, *e.g.* the HyperText Markup Language HTML. XML is classified as extensible since users can define their own tags, *cf.* HTML. Not only can XML tags be used to specify how to display the data, they may also be used to identify it. For example the pair of tags `<b> ...</b>` in HTML specifies that the enclosed text is displayed in bold font, whereas the pair `<message> ...</message>` in XML labels the enclosed text as a message. Below is some example XML data which might be used in a messaging application, say.

```
<message>
  <to> alice@pobox.com </to>
  <from> white.rabbit@adam.com.au </from>
  <subject> test </subject>
  <text> hello </text>
</message>
```

The tags identify the message in its entirety, the sender and destination addresses, the subject and the text of the message. Tags may also contain attribute-value pairs where the attribute names are followed by an equal sign and the attribute value. Furthermore, tags may be empty and contain no content. For example,

```
<message  to="alice@pobox.com"  from="white.rabbit@adam.com.au"  subject="test">
  <flag/>
  <text> hello </text>
</message>
```

Here the `message` start tag contains three attribute-value pairs with attribute names `to`, `from` and `subject` respectively. The empty tag `<flag/>` marks the message as noteworthy. All XML files begin with a prolog which contains a declaration that identifies the document as a XML document. The minimal prolog `<?xml  version="1.0"?>` may contain additional attribute-value pairs specifying the character set encoding or whether or not the document contains external references.

A XML schema - written in a schema language - specifies a vocabulary for the kinds of tags which can be included in a XML document and the valid arrangements of those tags. The schema usually features as part of the prolog, but it can be referred to externally or be split between prolog and one or more additional references. A XML parser checks whether a document conforms to its schema. XML Schema is one of several XML schema

languages. Another is DTD - derived from 'Document Type Definition' - which is native to the XML specification. Unlike other languages however, XML Schema is the first schema language to be recommended by the World Wide Web Consortium. More information regarding XML Schema can be found at [Sperberg-McQueen & Thompson 2007].

8.2 RDF, RDF Schema and SPARQL

RDF - derived from the 'Resource Description Framework' - is a language for representing information about resources in the World Wide Web [Manola & Miller 2004]. Statements about resources are made using a subject-predicate-object expression called a triple. The subject of a RDF statement denotes a resource in the form of a Uniform Resource Identifier (URI) reference. URIs are essentially generalised versions of Uniform Resource Locators (URLs). URLs are simply character strings which identify Web resources by describing their network locations, whereas a URI can be classified as a locator, a name identifying a resource without implying its location, or both. Essentially a URI can be created to refer to anything that needs to be referred to in a RDF statement. The predicate of a RDF statement denotes traits or aspects of the subject, and expresses a relationship between the subject and object. The object may either be a URI reference or a constant value represented by character strings. RDF represents triples as a graph of nodes (subjects and objects) and arcs (predicates). An arc is directed from subject node to object node. Nodes that are URI references are depicted as ellipses, whereas nodes representing constant values are boxes. As an example, consider the following graph adapted from [Manola & Miller 2004].

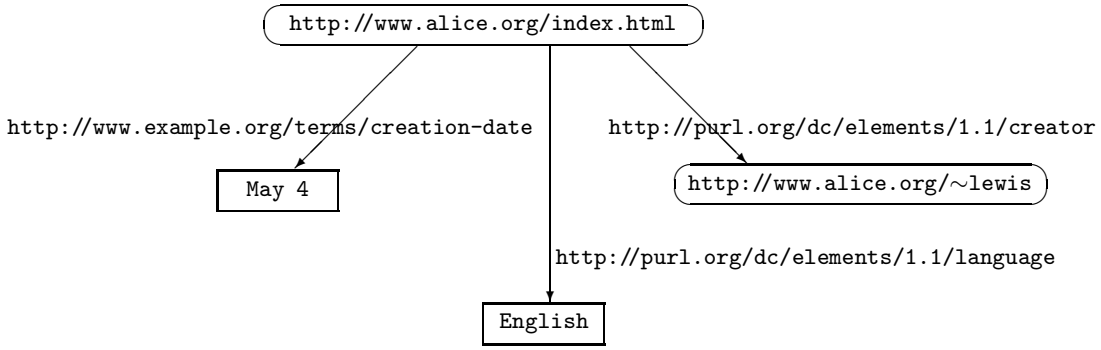


Figure 4: *RDF Graph*

This graph/(RDF statement) can be thought to represent the following three sentences.

- `http://www.alice.org/index.html` has a `creation-date` whose value is `May 4`.
- `http://www.alice.org/index.html` has a `language` whose value is `English`.
- `http://www.alice.org/index.html` has a `creator` whose value is the URI reference `http://www.alice.org/~lewis`

Each sentence corresponds to a RDF triple. The subject is identified by its URL `http://www.alice.org/index.html`. The predicates `creation-date`, `language` and `creator` of the statement are each identified by their respective URIs. The objects `May 4` and `English` are simply constant values, whereas the object related *via* the `creator` predicate is identified by its URI.

In order to represent RDF graphs in a machine-processable way, RDF uses XML. A specific notation called RDF/XML is used to write a RDF graph as an XML file. Note that this marking-up of RDF into XML is termed serialisation. There are a number of other serialisation notations such as Turtle, N3 and N-triples which we'll discuss shortly. First, we present our example above in RDF/XML. Note that each line in the example is numbered to help aid the explanation. Moreover, the example features qualified names which abbreviate URI references. A qualified name contains both a prefix that has been assigned to a URI and a local name separated by a colon. The full URI reference is formed by appending the local name to the URI assigned to the prefix. For example, if the prefix `rdf` is assigned to the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, then the qualified name `rdf:type` is shorthand for the URI reference `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`. Adapted from the RDF primer at [Manola & Miller 2004], we have the following XML file written in RDF/XML.

```
1.  <?xml   version="1.0"?>
2.  <rdf:RDF   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
3.             xmlns:exterms="http://www.alice.org/terms/"
4.             xmlns:dc="http://purl.org/dc/elements/1.1/">
5.    <rdf:Description   rdf:about="http://www.alice.org/index.html">
6.      <exterms:creation-date> May 4 </exterms:creation-date>
7.      <dc:language> English </dc:language>
8.      <dc:creator   rdf:resource="http://www.alice.org/∼lewis"/>
9.    </rdf:Description>
10. </rdf:RDF>
```

Figure 5: *RDF/XML*

Line 1 is simply the XML prolog. Line 2 begins with a `rdf:RDF` start tag. This signals that the subsequent enclosed content - from the attribute-value pair at line 2 to the closing tag `</rdf:RDF>` at line 10 - is intended to represent RDF. The attribute `xmlns` at line 2 identifies a XML name-space declaration. It assigns the prefix `rdf` to the URI `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, hence the subsequent content can then use qualified names - *e.g.* `rdf:Description` - as tags. Lines 3 and 4 specify two more namespace declarations for prefixes `exterms` and `dc`, and closes the `rdf:RDF` start tag at the end of line 4. Lines 5-9 capture the crux of the RDF statement. The `rdf:Description` start tag in line 5 indicates the start of a resource description. It moreover uses the `rdf:about` attribute to specify the URI reference of the subject of the statement. Lines 6-8 identify the predicates and objects of the statement. They use the qualified names `exterms:creation-date`, `dc:language` and `dc:creator` as tags. The content of the tags at lines 6 and 7 are the constant values (and objects of the RDF statement) `May 4` and `English`. Line 8 contains an empty `dc:creator` tag. Its content -

again another object of the statement - is another resource indicated by the `rdf:resource` attribute. The resource description is closed at line 9, and the RDF content is closed at line 10.

The serialisation notation Turtle - derived from 'Terse RDF Triple Language' - was designed with human readability in mind. It is a more compact and readable alternative to RDF/XML. An example illustrating this is the XML name-space declaration `<rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/">` which can be written in Turtle notation as: `@prefix dc:<http://purl.org/dc/elements/1.1/>`. Note that both have the same semantics, whereby the prefix `dc` is assigned to the URI reference `http://purl.org/dc/elements/1.1/`. Also, `<dc:language> English </dc:language>` can be represented in Turtle as: `dc:language "English"`. The notation N3 extends Turtle by providing features that go beyond the serialisation of RDF graphs, such as support for RDF rules. We refer the reader to [Berners-Lee 2000] for details. The simple line-based notation N-triples is a subset of Turtle [Grant & Beckett 2004]. Triples are listed on separate lines, with no abbreviations, *i.e.* with no qualified names. An example is the subject-predicate-object triple of our example Web page and the value of its language, which is written in the N-triple notation as

```
<http://www.alice.org/index.html> <http://purl.org/dc/elements/1.1/language> "English"
```

We refer the reader to [Decker et al. 2000, Manola & Miller 2004] for more information regarding RDF and its serialisation notations.

Just as a XML schema provides a vocabulary for the kinds of tags which may be used in a XML document, a RDF schema provides a similar facility for RDF. The RDF Schema language defines a vocabulary for RDF attributes as well as specifying the kinds of objects to which these attributes may be applied [Brickley & Guha 2004]. Essentially a schema provides a basic type system for use in RDF models. Importantly, it allows resources to be described as classes, instances of classes and subclasses of classes.

SPARQL - derived recursively from the 'SPARQL Protocol And RDF Query Language' - is a RDF query language which has been standardised by the World Wide Web Consortium. As discussed in [Prud'hommeaux & Seaborne 2007] SPARQL is based on triple patterns. These are essentially RDF triples where each subject, predicate and object may be a variable. A set of triple patterns is called a basic graph pattern. A basic graph pattern is said to match a RDF subgraph when RDF terms from that subgraph - *i.e.* URIs, constant values - may be substituted for the variables. A simple example is to find the language of our example webpage. We've already seen the following N-triple.

```
<http://www.alice.org/index.html> <http://purl.org/dc/elements/1.1/language> "English"
```

If we consider the RDF graph depicted in Figure 8.2, then this N-triple corresponds to the subgraph whereby the arrow labelled `http://purl.org/dc/elements/1.1/language` points from the ellipse representing the URI `http://www.alice.org/index.html` to the box representing the constant value `English`. We can pose the following SPARQL query over the entire RDF graph depicted in Figure 8.2.

```
SELECT  ?language
WHERE   {<http://www.alice.org/index.html> <http://purl.org/dc/elements/1.1/language>
                                                                    ?language.}
```

All SPARQL queries consist of (1) the `SELECT` clause which identifies the variables to appear in the query results, and (2) the `WHERE` clause which provides the basic graph pattern to match against the RDF graph. Our particular query will substitute the constant value `English` in the RDF subgraph/N-triple for the variable `?language` in the `WHERE` clause, and return the following answer.

language
"English"

We refer the reader to [Prud'hommeaux & Seaborne 2007] for more information regarding SPARQL.

8.3 OWL and SWRL

The Web Ontology Language OWL - as its name states - is an ontology language for the Web. It was designed to augment the facilities already provided by XML, RDF and RDF Schema for expressing semantic content on the Web. As discussed in [McGuinness & Harmelen 2004] OWL consists of three increasingly expressive sublanguages: OWL Lite, OWL DL and OWL Full. OWL DL derives its name from its relation to description logics; as mentioned in Section 6, it is based on the description language $\mathcal{SHOIN}(D)$. OWL Lite is a simple ontology language which allows users to develop classification hierarchies with simple constraints. OWL DL is more expressive than OWL Lite, yet remains decidable. OWL Full is more expressive still, but is not decidable. OWL Full is considered to be an extension of RDF, whereas OWL Lite and OWL DL are considered to be extensions of a subset of RDF.

In this section we'll give some very simple examples of OWL. To begin with, we'll look at a minimal OWL header.

```
<?xml   version="1.0"?>
<rdf:RDF   xmlns:mon="http://www.alice.org/monarchs#"
           xmlns:owl="http://www.w3.org/2002/07/owl#"
           xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <owl:Ontology   rdf:about="">
    <rdfs:comment> An example OWL ontology </rdfs:comment>
    <owl:imports   rdf:resource="http://www.alice.org/croquet"/>
    <rdfs:label> Monarchy Ontology </rdfs:label>
  </owl:Ontology>
</rdf:RDF>
```

Figure 6: *OWL Header*

A standard initial component of an OWL ontology is a set of XML namespace declarations. The first declaration identifies the URI of the document containing the current ontology with the prefix `mon`. The other namespace declarations are conventional and are

used to introduce the OWL, RDF and RDF Schema vocabularies respectively. The content between the `owl:Ontology` tags describes meta-data about the ontology stipulated by the `rdf:about` attribute. The standard value `""` refers to the URI of the document containing the current ontology. The `rdfs:comment` tags contain an annotation, whereas the empty `owl:imports` tag imports a `croquet` ontology into the current ontology. The `rdfs:label` tags contain a natural language label for the ontology. The header is closed with the tags `</owl:Ontology>` and `</rdf:RDF>`. The actual definitions that make up the ontology follow on from the header.

As discussed in [Smith, Welty & McGuinness 2004] much of an OWL ontology concerns classes, instances of classes, class individuals, and properties and relationships between these elements. An example of a user-defined class is given below.

```
<owl:Class   rdf:ID="RoyalMonarch"/>
```

The `rdf:ID="RoyalMonarch"` attribute-value pair assigns a name to a class. After this assignment `rdf:resource="#RoyalMonarch"` may be used to identify the URI of the `RoyalMonarch` class as `http://www.alice.org/monarchs#RoyalMonarch`. Note that each user-defined class is implicitly a subclass of `owl:Thing`. The empty class `owl:Nothing` is also defined. In order to define a subclass `Queen` we use the attribute `rdfs:SubClassOf`.

```
<owl:Class   rdf:ID="Queen">
  <rdfs:SubClassOf   rdf:resource="#RoyalMonarch"/>
</owl:Class>
```

In OWL, if `X` is a subclass of `Y`, then every instance of `X` is also an instance of `Y`. We can introduce an individual `QueenOfHearts` of the class `Queen` as follows.

```
<Queen   rdf:ID="QueenOfHearts">
```

An OWL object property is a binary relation which relates two class instances. We define the object property `obeysQueen` and specify its domain and range accordingly.

```
<owl:ObjectProperty   rdf:ID="obeysQueen">
  <rdfs:domain   rdf:resource="#Person"/>
  <rdfs:range   rdf:resource="#Queen"/>
</owl:ObjectProperty>
```

Here we assume we have previously defined the class `Person`. In OWL we may also define object properties as being transitive, symmetrical, functional, or the inverse of each other. (As an aside, the primary reason why OWL Full is undecidable is because it places no restriction on the transitivity of object properties.)

Our description thus far of the OWL language has only scratched the surface; we refer the reader to [Smith, Welty & McGuinness 2004] for a complete guide. As a Semantic Web technology, OWL is increasingly being implemented in a range of applications. Moreover, it is beginning to play a major role in the research of various reasoning techniques and language extensions. Recently an extension of OWL DL - based on the $\mathcal{SROIQ}(D)$ description language - has been proposed which extends OWL DL with a small set of features requested by users [Patel-Schneider & Horrocks 2007, Horrocks, Kutz & Sattler

2006]. OWL 1.1 has no official status yet, but a number of OWL reasoning tools - including Pellet, discussed below - have expressed a commitment to support the extension.

SWRL - the name derived from 'Semantic Web Rule Language' - has been proposed as the basic rules language for the Semantic Web [Horrocks et al. 2004]. It combines the sublanguages OWL DL and OWL Lite with RuleML, the Rule Markup Language. Note that reasoning using the full SWRL specification is undecidable. To solve this problem, a decidable subset of SWRL has been identified and is referred to as being DL-safe. We refer the reader to [Motik, Sattler & Studer 2005] for a discussion.

8.4 Implementations

Pellet is an OWL DL reasoner written in Java [Parsia & Sirin 2007, Sirin et al. 2007]. Originally developed at the University of Maryland and now commercially supported by Clarke & Parsia, Pellet implements a tableau-based satisfiability algorithm for the $\mathcal{SHOIN}(D)$ description language. It also supports reasoning over the new OWL 1.1 extension. Pellet's query engine is capable of answering conjuncted ABox queries posed in the SPARQL language. The tool supports reasoning with DL-safe rules encoded in SWRL, and allows incremental reasoning, whereby the ABox is incrementally updated over consecutive reasoning cycles.

Jena is a Java framework - *i.e.* Application Programming Interface (API) - for building Semantic Web applications. Developed at Hewlett Packard, Jena provides a programming environment for RDF and OWL, a SPARQL query engine, and a rule-based inference engine [McBride 2007]. In Jena, a RDF graph is represented as an instance of the class `Model`. In the abstract, this class instance contains a set of statement objects, each of which is a RDF triple. As described in [Dickinson 2007], `Model` comes with a variety of methods which allow a user to: read and write RDF documents in a variety of notations, *i.e.* RDF/XML, N-triples and N3; query models regarding the statements they contain; create and remove statements from models; and make 'statements about statements'. Jena also comes with its own rule language and a built-in inference engine. The tool also features a Reasoning API which can be used to implement a range of external inference engines or OWL reasoners.

Protégé - developed at the Stanford Center for Biomedical Informatics Research - is a Java-based ontology development environment [Musen, Noy & O'Connor 2007]. It supports two main ways of modelling ontologies. The Protégé-Frames editor allows users to build frame-based ontologies. Here an ontology consists of a set of classes organised in a subsumption hierarchy, a set of class instances, and a set of slots associated with each class which are used to describe their properties and relationships. The Protégé-OWL editor allows users to build OWL ontologies using classes, properties, individuals and SWRL rules. The editor also allows SPARQL questioning and answering. By way of its API, Protégé-OWL can be extended with various tools and applications. Moreover, external reasoning services may be accessed by way of a reasoning API.

9 Discussion

As discussed in [Brachman & Levesque 2004] the main problem faced by the KR&R community is this: although a highly expressive, formal knowledge representation language is desirable from a representation standpoint, it is *not* desirable from a reasoning standpoint. This is because reasoning procedures involving expressive representation languages are (in general) intractable, meaning that they often take an inordinate amount of time, or use excessive resources. Sometimes the procedures are even undecidable. There are two compromises we can make in regards to the intractability problem: (1) we can limit the expressiveness of the representation language, or (2) we can use an undecidable reasoner. Having to make such compromises is not the end of the world. Even limited representation languages can be extended in various ways to make them more useful in practice. (Note that determining which properties affect computational difficulty is still an open problem. Some progress has been made; for example in the field of description logics, much research has been conducted on which combinations of operators preserve tractability.) Moreover, undecidable reasoning algorithms are actually quite popular; they are considered good approximations of problem solving procedures.

In search of a knowledge representation and reasoning system for our information fusion task, we have surveyed the key areas of research in the KR&R field. For our information fusion task we require a knowledge representation language which combines the advantages of both natural and formal languages. It needs to be expressive enough to adequately represent whatever it is we want it to, without being ambiguous or context dependent. The language needs efficient inference procedures to form a complete knowledge representation system. Moreover, the language should be extensible; adding new constructs should not affect the system as a whole. Choosing a knowledge representation language boils down to this: how we represent knowledge depends on the inferences we want to make from that knowledge. We need to know what we want to *do* with our knowledge in order to assign it an appropriate representation language.

As far as our information fusion task goes, we want to represent and reason about information involving individuals and events. Much of the information we have will be incomplete. Individuals are likely to be described using attributes such as name, address, date-of-birth, eye-colour, known associates, *etc.* Events are likely to be described using attributes such as location, time, duration, outcomes, attendees, *etc.* We will want to make standard and (relatively) simple inferences about individuals, *i.e.* determining whether two individuals are the same person by matching their attribute values, or determining who an individual's known associates are. Other inferences are likely to be more complex, *i.e.* determining an individual's relationship to another, what their location is at the time of a particular event, whether two descriptions of events describe the same event, *etc.* Here more sophisticated reasoning will be required. There is an argument for using a hybrid knowledge representation system. Such a system would allow us to use efficient reasoning procedures - which determine the satisfiability of input formulae or check subsumption - along with more general first-order reasoning as the need arises. Of course we need to weigh-up using existing tools *vs.* building an in-house implementation.

Although first-order logic is a highly expressive knowledge representation language, a major drawback of the logic as a KR&R system for our information fusion task is its

undecidability. Moreover, most first-order automated theorem provers are not designed for large knowledge-based applications. Note there are some exceptions, for example KRHyper [Wernhard 2008]. Modal logics are gradually receiving more attention by the Artificial Intelligence community, but research in modal logics for knowledge representation still has a long way to go. A production rule (expert) system is viable as a KR&R system, but these systems are optimally suited for small, specific domains. To build an intelligence expert system we would require expert knowledge in pretty much everything. Frame systems are limited in their expressiveness, and moreover - in regards to knowledge representation - have been superceded by description logics. Semantic networks are great for taxonomies, but are not particularly suitable for our information fusion task. On a more positive note, description logics are currently very popular and are actively being researched. They are (in the most part) decidable and their open-world semantics would allow us to represent incomplete information. A further advantage is the availability of Semantic Web technologies such as OWL, SWRL and a number of reasoning tools. DL is still limited however; for our task, we'd need to look at a very expressive DL which might lose us decidability.

An important aspect we have not covered in this report is that knowledge is often affected by some degree of uncertainty. This can arise from unreliable information sources, or inaccurate, incomplete or out-of-date information. Moreover natural language can sometimes be inherently ambiguous or vague. In order to represent uncertain or vague knowledge, a number of KR&R methodologies have been developed, *e.g.* Probability Logic, Fuzzy Logic, Bayesian Networks, and Dempster-Shafer theory. It is highly likely that we will have to consider such methodologies in the future in regards to our information fusion task. In Probability Logic, the truth values of formulae are given as probabilities. In Fuzzy Logic, degrees of truth are represented by partial membership within sets. Using Bayesian Networks we can graphically represent a set of variables and their probabilistic interdependencies; whereas using Dempster-Shafer Theory we can combine separate pieces of information in order to calculate the probability of an event. We refer the reader to [Halpern 2005] for an in-depth discussion on these topics.

10 Conclusion

This report has surveyed many of the key principles underlying research in the field of knowledge representation and reasoning. We have provided an overview of a number of systems with different methodologies and emphases, logic-based or otherwise. The report represents an initial step in deciding upon a KR&R system for our information fusion task. It is future work to investigate in-depth the typical inferences we will wish to make. This will give us a better idea of the type of KR&R system we will require. We will also need to consider the availability of relevant tools and whether they may be adapted/extended for our task. The author is particularly interested in very expressive description logics and the formation of a (possibly undecidable) hybrid reasoning system. This will require further research.

References

Al-Halimi, R., Berwick, R. C., Burg, J., Chodorow, M. & Fellbaum, C. (1998) *Wordnet: An Electronic Lexical Database*, Bradford Books.

Armstrong, E. (2001) Working with XML. URL – `http://www2.informatik.hu-berlin.de/~xing/Lib/Docs/jaxp/docs/tutorial/`.

Baader, F. (2003) Description logic terminology, *in the Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, pp. 485–495.

Baader, F. & Nutt, W. (2003) Basic description logics, *in the Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, pp. 43–95.

Barwise, J. & Hammer, E. (1994) *What is a Logical System?*, Oxford University Press, chapter 3, Diagrams and the Concept of Logical System, pp. 73–106.

Berners-Lee, T. (2000) Notation 3: A readable language for data on the web. URL – `http://www.w3.org/DesignIssues/Notation3.html`.

Blackburn, P. & Bos, J. (2005) *Representation and Inference for Natural Language: A First Course in Computational Semantics*, CSLI.

Blackburn, P., Bos, J. & Striegnitz, K. (2006) *Learn Prolog Now!*, College Publications.

Borgida, A. (1996) On the relative expressiveness of description logics and predicate logics, *Artificial Intelligence* **82**(1-2), 353–367.

Brachman, R. J. (1977) *A Structural Paradigm for Representing Knowledge*, PhD thesis, Harvard University.

Brachman, R. J. & Levesque, H. (2004) *Knowledge Representation and Reasoning*, Elsevier.

Brachman, R. J. & Schmolze, J. G. (1985) An overview of the KL-ONE knowledge representation system, *Cognitive Science* **9**(2), 171–216.

Brickley, D. & Guha, R. (2004) RDF vocabulary description language 1.0: RDF Schema. URL – `http://www.w3.org/TR/rdf-schema/`.

Colmerauer, A. & Roussel, P. (1993) The birth of Prolog, *in proceedings of the 2nd ACM SIGPLAN conference on the History of Programming Languages*, ACM Press.

Cresswell, M. J. (2001) Modal logic, *in the Blackwell Guide to Philosophical Logic*, Blackwell, pp. 136–158.

Dau, F. (2003) *The Logic System of Concept Graphs with Negation and its Relationship with Predicate Logic*, Vol. LNAI 2892, Springer.

Davis, R., Shrobe, H. & Szolovits, P. (1993) What is a knowledge representation?, *AI Magazine* **14**(1), 17–33.

Decker, S., Harmelen, F. v., Broekstra, J., Erdmann, M., Fensel, D., Horrocks, I., Klein, M. & Melnik, S. (2000) The semantic web: The roles of XML and RDF, *IEEE Internet Computing* **4**(5), 63–74.

Delugach, H. & Menzel, C. (2007) Common logic standard. URL – `http://common-logic.org`.

Dickinson, I. (2007) Meet Jena, a semantic web platform for Java. URL – `http://www.devx.com/semantic/Article/34968/`.

Emerson, E. A. (1990) Temporal and modal logic, *in the Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics*, Elsevier, pp. 996–1072.

Ferrucci, D., Lally, A., Gruhl, D., Epstein, E., Schor, M., Murdock, W. & Frenkiel, A. (2006) *Towards an Interoperability Standard for Text and Multi-Modal Analytics*, Technical report, IBM RC24122.

Franconi, E. (2002) Description logic tutorial. URL – `http://www.inf.unibz.it/~franconi/dl/course/`.

Friedman-Hill, E. (2007) Jess: the rule engine for the Java platform. URL – `http://www.jessrules.com/`.

Genesereth, M. & Fikes, R. (1992) *Knowledge Interchange Format, Version 3.0*, Technical report, Logic-92-1, Stanford Logic Group, Stanford University.

Goré, R. (1999) Tableau methods for modal and temporal logics, *in the Handbook of Tableau Methods*, Kluwer, pp. 297–396.

Goré, R. (2003) *History and Philosophy of Science for African Undergraduates*, Vol. 3, Ghana University Press, chapter 45, An Introduction to Classical Propositional Logic: Syntax, Semantics, Sequents, pp. 597–643.

Grant, J. & Beckett, D. (2004) RDF test cases. URL – `http://www.w3.org/TR/rdf-testcases/`.

Halpern, J. Y. (2005) First-order modal logic, *in Reasoning About Uncertainty*, MIT Press, pp. 365–392.

Hayes, P. (1977) In defense of logic, *in proceedings of the 5th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 559–565.

Hayes, P. (1979) The logic of frames, *in Readings in Knowledge Representation*, Morgan Kaufmann, pp. 287–295.

Helbig, H. (2005) Knowledge representation with multilayered extended semantic networks: the MultiNet paradigm. URL – `http://pi7.fernuni-hagen.de/forschung/multinet/multinet_en.html`.

Herman, I. (2001) Semantic web activity statement. URL – `http://www.w3.org/2001/sw/Activity.html`.

Horrocks, I., Kutz, O. & Sattler, U. (2006) The even more irresistable SROIQ, *in proceedings of the 10th International Conference on the Principles of Knowledge Representation and Reasoning*, pp. 57–67.

Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B. & Dean, M. (2004) SWRL: A semantic web rule language combining OWL and RuleML. URL – `http://www.w3.org/Submission/SWRL/`.

Israel, D. J. (1983) The role of logic in knowledge representation, *IEEE Computer* **16**(10), 37–42.

Kremer, R. & Lukose, D. (1996) Knowledge engineering, part A: Knowledge representation. URL – `http://pages.cpsc.ucalgary.ca/~kremer/courses/CG/`.

Kripke, S. (1963) Semantical analysis of modal logic: Normal propositional calculi, *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* **9**, 67–96.

Lukasiewicz, T. (2007) *Probabilistic Description Logics for the Semantic Web*, Technical report, INFYS Research Report 1843-06-05, Technical University of Vienna, Austria.

Manola, F. & Miller, E. (2004) RDF primer. URL – `http://www.w3.org/TR/rdf-primer/`.

McBride, B. (2007) An introduction to RDF and the Jena RDF API. URL – `http://jena.sourceforge.net/tutorial/RDF_API/`.

McGuinness, D. & Harmelen, F. v. (2004) OWL web ontology language overview. URL – `http://www.w3.org/TR/owl-features/`.

Minsky, M. (1975) A framework for representing knowledge, *in the Psychology of Computer Vision*, McGraw-Hill, pp. 211–277.

Motik, B., Sattler, U. & Studer, R. (2005) Query answering for OWL-DL with rules, *Journal of Web Semantics: Science, Services and Agents on the World Wide Web* **3**(1), 41–60.

Musen, M., Noy, N. & O'Connor, M. (2007) Protégé ontology editor. URL – `http://protege.stanford.edu/`.

Nardi, D. & Brachman, R. J. (2003) An introduction to description logics, *in the Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, pp. 1–40.

Nivelle, H. d., Schmidt, R. & Hustadt, U. (2000) Resolution-based methods for modal logics, *Logic Journal of the Interest Group in Pure and Applied Logic (IGPL)* **8**(3), 265–292.

Øhrstrøm, P. (1997) C.S. Peirce and the quest for gamma graphs, *in proceedings of the 5th International Conference on Conceptual Structures*, Vol. Conceptual Structures: Fulfilling Peirce's Dream, LNAI 1257, Springer, pp. 357–370.

Parsia, B. & Sirin, E. (2007) Pellet. URL – `http://pellet.owldl.com/`.

Patel-Schneider, P. & Horrocks, I. (2007) OWL 1.1 web ontology language overview. URL – http://www.webont.org/owl/1.1/overview.html.

Pease, A. (2008) The suggested upper merged ontology. URL – http://www.ontologyportal.org.

Prud'hommeaux, E. & Seaborne, A. (2007) SPARQL query language for RDF. URL – http://www.w3.org/TR/rdf-sparql-query/.

Sattler, U., Calvanese, D. & Molitor, R. (2003) Relationships with other formalisms, *in the Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, pp. 137–177.

Schärfe, H. (2007) Online course in knowledge representation using conceptual graphs. URL – http://gda.utp.edu.co/traductor/Documentacion/SitiosWeb/www.humin f.aau.dk/cg/index.html.

Shortliffe, E. (1981) Consultation systems for physicians: The role of artificial intelligence techniques, *in Readings in Artificial Intelligence*, Tioga Publishing Company, pp. 323–333.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A. & Katz, Y. (2007) Pellet: A pratical OWL DL reasoner, *Journal of Web Semantics* **5**(2).

Smith, M. K., Welty, C. & McGuinness, D. (2004) OWL web ontology language guide. URL – http://www.w3.org/TR/owl-guide/.

Sowa, J. F. (1984) *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley.

Sowa, J. F. (2000) *Knowledge Representation: Logical, Philosophical and Computational Foundations*, Brooks/Cole.

Sowa, J. F. (2001) Conceptual graphs: Working draft of the proposed ISO standard. URL – http://www.jfsowa.com/cg/cgstand.htm.

Sowa, J. F. (2003) Existential graphs: Notes on MS 514 by Charles Sanders Peirce. URL – http://jfsowa.com/peirce/ms514.htm.

Sperberg-McQueen, C. & Thompson, H. (2007) XML Schema. URL – http://www.w3.org/TR/2007/WD-xmlschema11-1-20070830/.

Vossen, P. & Fellbaum, C. (2007) The global WordNet association. URL – http://www.globalwordnet.org/.

Wernhard, C. (2008) KRHyper. URL – http://www.uni-koblenz.de/~wernhard/krhyper/.

Zalta, E. N. (1995) Basic concepts in modal logic. URL – http://mally.stanford.edu.

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA	1. CAVEAT/PRIVACY MARKING

2. TITLE	3. SECURITY CLASSIFICATION
Survey of Knowledge Representation and Reasoning Systems	Document (U) Title (U) Abstract (U)
4. AUTHOR	5. CORPORATE AUTHOR
Kerry Trentelman	Defence Science and Technology Organisation PO Box 1500 Edinburgh, South Australia 5111, Australia

6a. DSTO NUMBER	6b. AR NUMBER	6c. TYPE OF REPORT	7. DOCUMENT DATE
DSTO–TR–2324	AR–014–588	Technical Report	July 2009

8. FILE NUMBER	9. TASK NUMBER	10. SPONSOR	11. No. OF PAGES	12. No. OF REFS
2009/1059398/1	NS07/201	Exec Dir CTSTC	49	65

13. URL OF ELECTRONIC VERSION	14. RELEASE AUTHORITY
http://www.dsto.defence.gov.au/corporate/reports/DSTO–TR–2324.pdf	Chief, Command, Control, Communications and Intelligence Division

15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT

Approved for Public Release

OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111

16. DELIBERATE ANNOUNCEMENT

No Limitations

17. CITATION IN OTHER DOCUMENTS

No Limitations

18. DSTO RESEARCH LIBRARY THESAURUS

Artificial Intelligence	Logic
Expert systems	Reasoning
Knowledge representation	Semantic web

19. ABSTRACT

As part of the information fusion task we wish to automatically fuse information derived from the text extraction process with data from a structured knowledge base. This process will involve resolving, aggregating, integrating and abstracting information - *via* the methodologies of Knowledge Representation and Reasoning - into a single comprehensive description of an individual or event. This report surveys the key principles underlying research in the field of Knowledge Representation and Reasoning. It represents an initial step in deciding upon a Knowledge Representation and Reasoning system for our information fusion task.