Chapter 2

Background

This chapter presents the background information of the various languages, notations, techniques and tools that are involved in this thesis. It is divided into five parts. In Section 2.1, we give a brief account of Semantic Web languages and tools. Following that, Section 2.2 is devoted to the introduction to the Semantic Web services ontology OWL-S, an OWL ontology that defines a set of core vocabularies for describing Web services. In Section 2.3, we briefly introduce the formal languages Z and Alloy and their tool support Z/EVES and Alloy Analyzer. Finally, institutions and institution morphisms are briefly covered in Section 2.4.

2.1 The Semantic Web – Languages & Tools

Proposed by Tim Berners-Lee et al., the Semantic Web [8] is a vision of next generation of the Web. The current World Wide Web is designed mainly for human consumption. It is believed that in the future, the Web is also ready for intelligent software agents and it will be truly ubiquitous. Software agents will reside in, for

example, household appliances (which can also be part of the Web), and will be able to understand the meaning of information on the Web and undertake tasks without human's supervision. To sum up, in the Semantic Web, software agents will be able to autonomously and cooperatively understand, process and aggregate Web resources, which include not only static data, but also dynamic Web services.

Semantic Web ontologies give precise and non-ambiguous *meaning* to Web resources, enabling software agents to understand them. An ontology is a specification of a conceptualization [34]. It is a description of the concepts and relationships for a particular application domain. Ontologies can be used by software agents to precisely categorize and deduce knowledge.

Languages in the Semantic Web

Ontology languages are the building blocks of the Semantic Web. As briefly mentioned in Chapter 1, the development of ontology languages takes a layered approach. Depicted in Fig. 1.1, the Semantic Web languages are constructed on top of mature languages and standards such as the XML [108], Unicode and Uniform Resource Identifier (URI) [7]. In the rest of this section, we briefly present some important languages in the Semantic Web.

The Resource Description Framework (RDF) [68] is a model of metadata that defines a mechanism for describing resources and makes no assumptions about a particular application domain. RDF allows structured and semi-structured data to be mixed and shared across applications. XML describes documents, whereas RDF is a framework for metadata: it describes actual things. RDF provides a simple *triples* structure to make *statements* about Web resources. Each triple is of the form $\langle subject\ predicate\ object \rangle$, where *subject* is the resource we are interested in, *predicate* specifies the property or characteristic of the subject and *object* states the value of

the property. Besides this basic structure, a set of basic vocabularies are defined to describe RDF ontologies. This set includes vocabularies for defining and referencing RDF resources, declaring containers such as bags, lists, and collections. It also has a formal semantics that defines the interpretation of the vocabularies, the entailment between RDF graphs, etc.

RDF Schema (RDFS) [17] defines additional language constructs for RDF ontologies. It adds considerable expressivity to RDF by enabling one to group Web resources into classes, to denote the domain and range of a property, to state the subsumption relationship between classes and properties, etc.

RDF Schema can be considered as the first ontology language for the Semantic Web. However, RDF and RDFS have a number of disadvantages. For instance, in order for agents to understand Web resources unambiguously, it is necessary that these resources are strictly structured. This requirement is relaxed by RDF to allow for greater flexibility. Also, RDF Schema does not contain all modeling primitives users desired.

In RDF, RDF Schema and subsequent ontology languages, Web resources are referenced using full , URI references. It consists of a URI prefix (a namespace) and the name of the resource, separated by a separator "#". RDF also defines a shorthand form for convenience. In this form, the full URI representing the resource is given an XML qualified name, containing a prefix that is assigned to the namespace URI, the local name (which is the name of the resource), separated by a colon (:). A number of qualified name prefixes have been predefined in the Semantic Web domain. These are summarized in Table 2.1.

With the above mapping between prefixes and full namespace URIs, a long URI reference can be shortened. For example, the full URI reference for RDFS class is `http://www.w3.org/2000/01/rdf-schema#Class`. With the above representation,

Table 2.1: Predefined Qualified Name Prefixes

Prefix	Namespace URI
xsd:	http://www.w3.org/2001/XMLSchema#
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
daml:	http://www.w3.org/2001/10/daml+oil#
owl:	http://www.w3.org/2002/07/owl#
swrl:	http://www.w3.org/2003/11/swrl#

it can be shortened to `rdfs:Class`.

The DARPA Agent Markup Language (DAML) is built on top of RDF Schema, but with a much richer set of language constructs to express class and property relationships and more refined support for data types. DAML project combined effort with the Ontology Inference Layer (OIL) [13] project and it is now referred to as DAML+OIL [101]. Being semantically equivalent to the expressive description logic $\mathcal{SHIQ}$ [50], the other major advantage of DAML+OIL over RDFS is the ability to define new classes and properties by defining restrictions on existing classes and properties. This enhances ontology structure and facilitates ontology reuse.

The main ingredients of DAML+OIL can be categorized into three types: objects, classes and properties, with data types supplying concrete values. The Object domain consists of objects (individuals) that are members of DAML+OIL or RDFS classes. Classes are the focus of DAML+OIL and they are elements of `daml:Class`, a sub class of `rdfs:Class`. DAML+OIL defines a number of built-in properties. They serve a number of purposes, which can be briefly summarized below.

- Some of the properties are used to relate two classes to define certain relationship between them. For example, the property `daml:disjointWith` is used to denote the disjointness of two classes.
- Some properties are used to construct classes from a list of classes or individ-

uals. For example, the property `daml:unionOf` relates a `daml:Class` X and a `daml:List` Y of classes such that the instances of X is the union of all the instances of classes in Y. The property `daml:disjointUnionOf` is similar, with an additional constraint that the classes in the list Y are mutually disjoint.

- Some properties are used to define new classes by constructing "restrictions", which are (anonymous) classes that can be linked to other properties or cardinality constraints.

  For example, the built-in property `daml:toClass` can be used to define the class of all objects for whom the values of property all belong to the class expression. It can be used to define, for instance, a restriction whose instances eats only Animals, as shown below.

```
<daml:Restriction>
   <daml:onProperty rdf:resource="#eats"/>
   <daml:toClass rdf:resource="#Animals"/>
</daml:Restriction>
```

  In the above example, the restriction is defined on the property `eats` and class `Animals`. This restriction can be used to define a class `Carnivores` by making it a sub class of this restriction.

  The cardinality properties define restrictions each of whose instances has exactly, at least or at most `n` distinct property values.

  The following DAML+OIL fragment defines a restriction, each of whose instances has *exactly one* nationality.

```
<daml:Restriction>
   <daml:cardinality>1</daml:cardinality>
   <daml:onProperty rdf:resource="#natitonality"/>
</daml:Restriction>
```

- Finally, some built-in properties are used to define or relate other properties. For example, the property `daml:samePropertyAs` asserts that the two properties it

relates are actually equivalent, meaning that their property extensions (the pair of objects they relate) are actually the same.

In 2003, the W3C published a new ontology language, the Web Ontology Language (OWL) [69] to replace DAML+OIL. Based on DAML+OIL, OWL is a suite of languages consisting of three species: Lite, DL and Full, with increasing expressiveness.

The three sublanguages are meant for user groups with different requirements of expressiveness and decidability. OWL Lite is the least expressive sublanguage, obtained by imposing restrictions on the usage of OWL Full language constructs. OWL DL is more expressive than Lite but is also a subset of OWL Full.

OWL Lite and DL are decidable whereas OWL Full is not. Simplistically speaking, an OWL Lite or DL ontology is an OWL Full ontology with some constraints added. These constraints include, for example, in OWL Lite, cardinality constraints can only be 0 or 1; mutual disjointness among individuals, classes, properties, data types, etc., in OWL Lite and DL ontologies. DAML+OIL is most comparable to OWL DL, which is a notational variance of description logic $\mathcal{SHOIN}$(D) [49].

The following OWL DL fragment shows the definition of carnivores in an animal-plant ontology. It defines an OWL class `Carnivores` that is a sub class of `Animals`. It is also a sub class of an anonymous class that only `eats Animals` (the allValuesFrom restriction). Note that the built-in DAML+OIL property `toClass` is renamed in OWL to `allValuesFrom`.

```
<owl:Class rdf:about="#carnivore">
   <rdfs:subClassOf>
      <owl:Class rdf:ID="animal"/>
   </rdfs:subClassOf>
   <rdfs:subClassOf><owl:Restriction><owl:onProperty>
      <owl:ObjectProperty rdf:ID="eats"/></owl:onProperty>
      <owl:allValuesFrom>
         <owl:Class rdf:about="#animal"/>
      </owl:allValuesFrom>
   </owl:Restriction></rdfs:subClassOf>
</owl:Class>
```

For any DAML+OIL or OWL ontology there are three types of core inference problems, namely concept (class) subsumption, concept consistency and instantiation reasoning. Concept subsumption checks if a concept subsumes another concept; concept consistency checks if a concept is meaningful with respect to the ontology, and property instantiation checks whether a given individual is an instance of a class. All the above inference problems can be checked by mature tableau algorithms for description logics in full automation.

The consistency of ontologies is essential to the proper functioning of agents. For example, we can imagine how chaotic it can be if an online marriage registry agent allows a person already married to register for marriage again. This could happen if the marriage ontology does not constrain that a person can only have at most one spouse. A consistent ontology satisfies the following two criteria: *realization*, that every class has at least one instance and *retrieval*, that every individual is an instance of some class [74]. Hence, the ontology consistency problem (and actually all the other types of inference problems) can be reduced to the concept consistency problem above.

Although the design of OWL has taken into consideration the different expressivity needs of various user groups, it is still not powerful enough as only relatively simple relationships can be expressed: such as class and property membership, individual (in)equalities, etc. The main reason for these limitations is that although OWL provides relatively rich language constructs for describing class relationships, it does not provide enough language primitives for describing properties. For example, properties in OWL cannot be composed to construct complex properties.

These limitations have been recognized by a number of researchers and in 2004, Horrocks and Patel-Schneider proposed a rules extension to OWL DL. The new language is called OWL Rules Language (ORL) [47] and it is syntactically and semantically

coherent to OWL. By incorporating Horn clause rules into OWL and making rules part of OWL axioms, which are used to construct classes and properties, ORL can express more complex properties. ORL is now known as SWRL [48], with some sets of built-ins for handling data type, such as numbers, booleans, strings, date & time, etc.

The major extensions of SWRL over OWL DL include Horn style rules and (universally quantified) variable declaration. For presentation and brevity purposes, the rules are in the form of `antecedent` $\rightarrow$ `consequent`, where both antecedent and consequent are conjunctions of the following kinds of atoms: class membership, property membership, individual (in)equalities and built-ins. Informally, a rule means that if the antecedent holds, the consequent must also hold. Moreover, an empty antecedent is treated as trivially true and an empty consequent is treated as trivially false. In SWRL, variables are prefixed with a question mark (?). A simple example rule states that if $?b$ is a parent of $?a$ and $?c$ is a brother of $?b$, then $?c$ is an uncle of $?a$, where $?a$, $?b$ and $?c$ are variable names.

$$\texttt{hasParent}(?a, ?b) \land \texttt{hasBrother}(?b, ?c) \rightarrow \texttt{hasUncle}(?a, ?c)$$

SWRL extends the expressivity of OWL by providing more support for describing and composing properties as shown in the previous example. It has been shown to be non-decidable. However, it is still not as expressive as Z. As one of the main motivations of the rules extension is to infer knowledge not present in the ontology, disjunction and negation are not allowed in SWRL. It also does not support explicit quantification over rules. As we stated above, these design constraints hinder expressing certain properties.

In view of this, Patel-Shneider proposed the language SWRL FOL [9] as a step further towards first-order logic. On top of SWRL, it adds logical connectors such as 'and',

'or', 'negation', 'implication', and 'existential' and 'universal' quantification.

The ontology languages DAML+OIL and OWL are based on description logics, for which highly optimized algorithms for solving concept consistency problems exist. However, OWL has also been criticized for a number of reasons [56], such as the inappropriate layering on top of RDFS; unnaturalness of certain modeling decisions; inefficiency of query answering mechanisms; the lack of distinction between restrictions and constraints, etc. To overcome these disadvantages, the OWL$^-$ [56] suite of languages were proposed. OWL$^-$ also consists of three sublanguages: OWL Lite$^-$, DL$^-$ and Full$^-$, where OWL Lite$^-$ and DL$^-$ are strict subsets of the respective OWL species. OWL DL$^-$ is an extension of OWL Lite$^-$ and OWL Full$^-$ is an extension of OWL DL$^-$ towards OWL Full.

The semantics of OWL$^-$ languages are based on logic programming. OWL Lite$^-$ and DL$^-$ are constructed in such a way that they can be directly translated into Datalog programs. Hence mature techniques in the deductive databases in query answering and rule extensions can be borrowed.

An extension to OWL$^-$, the OWL Flight [20], has also been proposed. It adds a number of features on top of OWL$^-$, such as constraints and local closed-world assumption.

The Web Rule Language (WRL) [1] is a proposal of a rule-based ontology language. Based on deductive databases and logic programming, WRL is designed to be complementary to OWL which is strong at checking subsumption relationships among concepts. WRL focuses on checking instance data, the specification of, and reasoning about arbitrary rules. A new layering of Semantic Web ontology languages is also proposed [19], as shown in Fig. 2.1. Moreover, WRL assumes a "Closed World Assumption", whereas OWL and SWRL assume an "Open World Assumption".

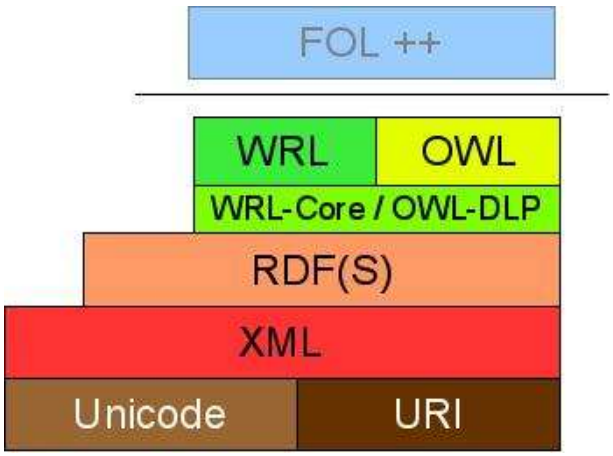

Figure 2.1: A newly proposed layering of the Semantic Web

There also exist other rules extensions besides the ones mentioned above. The Semantic Web Services Language (SWSL) [2] has been developed under the Semantic Web Services Initiative (SWSI)[1] framework. It is a logic-based language for specifying formal characterizations of Web service concepts and descriptions of individual services. However, SWSL is domain-independent and it does not contain any constructs customized to Web services. SWSL has a layered structure. Unlike OWL, the layers of SWSL are not organized according to expressivity. Rather, the SWSL layers are orthogonal to each other and each introduces new features that enhance the modeling power of the language. Moreover, these layers can be implemented together or in any arbitrary combination so that users can implement the reasoning service according to features required. SWSL includes two sublanguages: SWSL-FOL, a full first-order logic language, which is used to specify the service ontology (SWSO), and SWSL-Rules, a rule-based sublanguage, which can be used both as a specification and an implementation language.

[1] cf. http://www.swsi.org/

Recently, the Rule Interchange Format (RIF) working group [2] has been formed by the W3C with the aim to producing a "standard means for exchanging rules on the Web".

Tools in the Semantic Web

Besides ontology languages, we also witness the growth of ontology tools in the recent years. Various tools have been built to facilitate the diversified range of ontology development tasks, including creation, management, versioning, merging, querying, verification, etc. Here we briefly survey a few. An extensive survey was provided in [77].

Cwm (Closed world machine) [96] is a general-purpose data processor for the SW. Implemented in Python and command-line based, it is a forward chaining reasoner for RDF.

Triple [87] is an RDF query, inference and transformation language. It does not have a built-in semantics for RDF Schema, allowing semantics of languages to be defined with rules on top of RDF. This feature of Triple facilitates data aggregation as user can perform RDF reasoning and transformation under different semantics. The Triple tool supports DAML+OIL through external DAML+OIL reasoners such as FaCT and RACER.

Fast **C**lassification of **T**erminologies (FaCT) [45], developed at University of Manchester, is a TBox (terminology Box, concept-level) reasoner that supports automated concept-level reasoning, namely class subsumption and consistency reasoning. It does not support ABox (assertion Box, instance-level) reasoning. FaCT implements a reasoner for the description logic $\mathcal{SHIQ}$ [50]. It is implemented in Common Lisp and comes with a

[2] cf. http://www.w3.org/2005/rules/.

FaCT server, which can be accessed across network via its CORBA interface. Given a DAML+OIL/OWL ontology, it can classify the ontology (performs subsumption reasoning) to reduce redundancy and detects any inconsistency within it.

Recently a new version, the FaCT++ [44] system was released. It is an OWL Lite reasoner and introduced some new optimization techniques.

RACER, the **R**enamed **A**Box and **C**oncept **E**xpression **R**easoner [36], implements a TBox and ABox reasoner for the description logic $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D})^-$ [35]. It can be regarded as (a) a SW inference engine, (b) a description logic reasoning system capable of both TBox and ABox reasoning and (c) a prover for modal logic Km. In the SW domain, RACER's functionalities include creating, maintaining and deleting ontologies, concepts, roles and individuals; querying, retrieving and evaluating the knowledge base, etc. It supports RDF, DAML+OIL and OWL. The RACER system has recently been commercialized and it is now known as RacerPro[3].

Both FaCT (FaCT++) and RACER (RacerPro) perform their functions in full automation, which means by "pushing a button", these tools return a definitive answer without intermediate steps.

OilEd [4] is a visual DAML+OIL and OWL ontology editor developed by the University of Manchester. In OilEd, users can create new classes/properties, relate them using restrictions, view the hierarchy of classes and create instances of classes. Protégé [30] is a system for developing knowledge-based systems developed at Stanford University. It is an open-source, Java-based Semantic Web ontology editor that provides an extensible architecture, allowing users to create customized applications. In particular, the Protégé-OWL plugin [57] enables editing OWL ontologies and connecting to DIG [5]-compliant reasoning engines such as RACER [36] and FaCT++ [44]

[3] cf. http://www.racer-systems.com/

to perform tasks such as automated consistency checking and ontology classification.

Both of the above two editors support reasoners that conform to the DIG interface [5]. Next we briefly introduce a few that is relevant to the thesis.

2.2 Semantic Web Services Ontology OWL-S

Web Services [4] are a W3C coordinated effort to define a set of open and industry-supported specifications to provide a standard way of coordination between different software applications in a variety of environments. A Web service is defined as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL [14]). Other systems interact with the Web service in a manner prescribed by its description using SOAP [110] messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [10].

The various specifications in the Web services domains are all based on XML, making information processing and interchange easier. However, as XML Schema only defines the syntax of a document, it is hard for software agents to understand the *semantics* of a Web service described using these specifications. A language that is both syntactically well-formed and semantical is therefore desirable.

As introduced in the previous section, the Semantic Web [8] is an envisioned extension of the current Web where resources are given machine-understandable, unambiguous meaning so that software agents can cooperate to accomplish complex tasks without human supervision.

[4] cf. http://www.w3.org/2002/ws/

OWL Services (OWL-S) [95] is a Web services ontology in OWL DL. It supplies Web service producers/consumers with a core set of markup language constructs for describing the properties and capabilities of their Web services in an unambiguous, computer-interpretable form. OWL-S was expected to enable the tasks of "automatic Web service discovery", "automatic Web service invocation" and "automatic Web service composition and inter-operation". OWL-S consists of three essential types of knowledge about a service: the profile, the process model and the grounding. Figure 2.2 shows the high-level architecture of an OWL-S ontology.

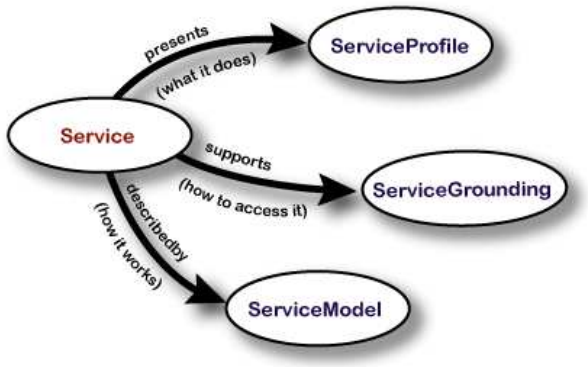

Figure 2.2: Architecture of the OWL-S ontology

A Web service consists of mainly three ingredients, a ServiceProfile, a ServiceGrounding and a ServiceModel. A ServiceProfile tells what the service does. It is the primary construct by which a service is advertised, discovered and selected. The ServiceGrounding tells how the service is used. It specifies how an agent can access a service by specifying, for example, communication protocol, message format, port numbers, etc.. The primary concern of our work in this paper is the OWL-S ServiceModel (also called process model), which tells how the service works. Thus, the OWL class Service is describedBy a ServiceModel. It includes information about the service's inputs, outputs, preconditions and effects. It also shows the component processes of a complex process and how the control flows between the components.

The OWL-S process model is intended to provide a basis for specifying the behaviors of a wide array of services. There are two chief components of an OWL-S process model – the process, and process control model. The process describes a Web Service in terms of its input, output, precondition, effects and, where appropriate, its component subprocesses. The process model enables planning, composition and agent/service inter-operation. The process control model – which describes the control flow of a composite process and shows which of various inputs of the composite process are accepted by which of its sub-processes – allows agents to monitor the execution of a service request. The constructs to specify the control flow within a process model include Sequence, Split, Split+Join, If-Then-Else, Repeat-While and Repeat-Until. The full list of control constructs in OWL-S and its semantics can be found in Chapter 7 and in the latest version of OWL-S [95].

2.3 Z & Alloy – Languages & Tools

The verification of Semantic Web ontologies to be presented in the following chapters involves the use of formal languages. In this section, we briefly introduce these languages, namely Z and Alloy, and their respective proof tool support.

2.3.1 Z

Z [107, 89] is a well-studied formalism based on ZF set theory and first-order predicate logic. Its formal semantics [106] and elegant modeling style encouraged an object-oriented extension, the Object-Z [28], and subsequently the Timed Communicating Object-Z (TCOZ) [67]. These additions greatly expand the expressivity of Z-family languages.

Z is specially suited to model system data and states. Z defines a number of language constructs including given type, abbreviation type, axiomatic definition, generic definition, state and operation schema definitions, etc. Besides, Z also defines a mathematical library, the *toolkit*, which gives definitions of commonly used concepts, symbols and operators, such as sets, set union, intersection, natural numbers, sequences, functions, relations, bags, etc.

Declarations

Z is a strictly-typed specification language. In Z, a name must be declared before it is referenced. Moreover, properties of systems being specified are stated using Z predicates. Hence, declarations and predicates are the building blocks of Z specifications.

The basic form of Z declarations is $x : A$, where x is the newly introduced variable of the type A. Moreover, this type A, which must be a set itself, should be defined previously too. A variable declared is either global or local. A global variable is visible from the point of declaration to the end of specification. A local variable's scope is the current enclosing environment. Interested readers may refer to [106, 88] for details.

Predicates

As in first-order logic, predicates in Z are Boolean-valued statements over a number of subjects. Z predicates allow the forms:

Equality & membership The basic Z predicates are equalities $=$ and membership relationships $\in$. For example, the predicate $x \in \mathbb{N}$ states that variable x is a member of natural numbers $\mathbb{N}$.

Set relationship operators such as subset can be derived using set membership. In general, the subset relationship $A \subseteq B$ can be expressed as $A \in \mathbb{P}\, B$, where $\mathbb{P}$ is the powerset symbol. The expression $\mathbb{P}\, B$ denotes all the sets that are subsets of B.

Propositional connectives These include the usual connectives in the propositional logic, namely $\neg$, $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$. They are used to connect simpler predicates to construct complex ones.

Quantifiers Based on first-order logic, Z also allows quantifiers in predicates. These include the universal quantifier $\forall$, the existential quantifier $\exists$ and the unique existential quantifier $\exists_1$. The predicate $\exists_1\, S \bullet P$ is true if there exists only one way of value assignment for the variables in S.

Note that the $\bullet$ symbol denotes "such that".

Let expressions The **let** expression constructs local definitions in a predicate. For example, in the predicate $\mathbf{let}\, x_1 == E_1;\ \ldots;\ x_n == E_n \bullet P$, the scope of variables $x_1, \ldots, x_n$ extends to the predicate P, but not into the bodies of expression $E_1, \ldots, E_n$.

The semantics of the **let** operator can be summarized as follows.

$$(\mathbf{let}\, x_1 == E_1;\ \ldots;\ x_n == E_n \bullet P) \\ \Leftrightarrow (\exists_1\, x_1 : t_1;\ \ldots;\ x_n : t_n \mid x_1 = E_1;\ \ldots;\ x_n = E_n \bullet P)$$

Note that the vertical bar | denotes the conditions that the expression in front of it must satisfy.

Relations Z also allows relation symbols to be used as predicates. The abstract syntax is defined as follows.

```
Predicate ::= Expression Rel Expression Rel ... Rel Expression
            | Pre-Rel Expression
```

Treated as predicates, relations denote relational memberships. For example, for a binary relation R, the predicate $E_1 \ R \ E_2$ denotes the membership predicate $(E_1, E_2) \in R$. The predicate $R \ E$, where R is a unary prefix symbol, denotes $E \in R$.

For the general form of chain of relations $E_1 \ R_1 \ E_2 \ R_2 \ E_3 \ R_3 \ldots E_{n-1} \ R_{n-1} \ E_n$, it is equivalent to the conjunction of individual relation predicate $E_1 \ R_1 \ E_2 \wedge E_2 \ R_2 \ E_3 \wedge \ldots \wedge E_{n-1} \ R_{n-1} \ E_n$.

Essential Language Constructs

In this subsection, we give a brief introduction to the more high-level Z language constructs relevant to this thesis. A more detailed introduction can be found in Appendix A.

***Given type*:**

A given type introduces uninterpreted basic types, which are treated as sets in Z. For example:

$[Resource]$

introduces one given type *Resource*, which is a set.

***Axiomatic definition*:**

An axiomatic definition defines global variables, and optionally constrains their values using predicates. These global variables cannot be globally redefined. For example, the following axiomatic definition defines two variables *Class* and *Property* as subsets of *Resource*. Furthermore, we assert that these two sets are mutually disjoint (their intersection is an empty set).

$$\begin{array}{|l} Class : \mathbb{P}\, Resource \\ Property : \mathbb{P}\, Resource \\ \hline Class \cap Property = \emptyset \end{array}$$

***Generic Definitions*:**

A generic definition is a generic form of axiomatic definition, parameterized by a formal parameter, a set X.

For example, in OWL DL, a datatype property relates some individuals to values of some data type. The mapping of such properties can be modeled by the following generic definition *sub_valD*. Note that in this definition the predicate part is empty.

$$\begin{array}{|l} [X] \\ \hline sub\_valD : DatatypeProperty \rightarrow (Individual \leftrightarrow X) \\ \hline \end{array}$$

***Constraints*:**

A constraint (predicate) constrains values of global variables that have been declared previously. For example, the following predicate states that the cardinality of the set is 2, implying that the two set members, which are both previously defined, are actually distinct.

$$\#\{PLAN\_P3\_P6\_P1, PLAN\_P3\_P6\} = 2$$

Ontology languages such as DAML+OIL and OWL are based on description logics, which are well known to be a subset of first-order logic [58]. Z, on the other hand, embraces expressivity from both first-order logic and schema calculus. Hence, Z is by nature more expressive than these languages. It is able to capture more complex properties pertaining to an ontology than ontology languages can.

Z/EVES [84] is an interactive system for composing, checking, and analyzing Z specifications. It supports the analysis of Z specifications in a number of ways: syntax and

type checking, schema expansion, precondition calculation, domain checking, general theorem proving, etc.

In Z/EVES, properties about a specification can be specified as *theorems*. These properties include facts and facts that one hopes to be facts. By proving theorems of a particular specification, we gain more confidence about its correctness.

The abstract syntax of theorems is defined as follows [71].

theorem ::= `\begin`[para-opt]{`theorem`}{[usage] theorem-name}[gen-formals]
predicate
[`\proof`
command sep ... sep command]
`\end{theorem}`

In the above abstract syntax, the keyword "para-opt" has two options: `disabled` or `enabled`, which indicate whether the theorem is to be automatically used by the theorem prover. The "gen-formals" keyword is an optional list of formal parameters appearing in the definition of the theorem.

The keyword "usage" have a number of options and it indicates the type of the theorem and consequently how it is to be used by Z/EVES. The options for this keyword are categorized as follows.

Facts The usage `axiom` indicates that the theorem is to be used by Z/EVES as a fact.

Rewrite rules The usage `rule` specifies that a theorem is to be used as a rewrite rule. Put it simply, a rewrite rule is a Z predicate, in the form of either a universal quantification, a logical implication, equivalence or an expression equality. If

such a theorem is used, Z/EVES will replace the left-hand side of the predicate or expression by its right-hand side during reduction and rewriting.

Forward rules The usage `frule` specifies that a theorem is to be used as a forward rule, which is in the form of an implication from a schema reference to a list of conjuncted predicates. A forward rule can be fired during simplification and it used to introduce predicates to Z/EVES.

Assumption rules The usage `grule` specifies that a theorem is to be used as an assumption rule. As its name suggests, an assumption rule is used to make Z/EVES assume some predicates. It can be used to introduce type information and inequalities into the proof context.

For example, the following theorem is a `disabled` assumption rule that states if a resource x is a member of *Class*, then it can be assumed that it is not a member of *Property*. Note that in theorems, variables can be used without declaration.

theorem disabled grule classPropertyDisjointRule
$x \in Class \Rightarrow x \notin Property$

In the ISO standard Z [52] and Z/EVES, Z specifications are organized into *sections* to improve specification clarity and reuse. The built-in section `toolkit`, as introduced above, defines basic constants and operators. Specifications are built hierarchically by including existing sections as their parents.

2.3.2 Alloy

Alloy [54] is a structural modeling language emphasizing on automated reasoning support. It treats relations as first-class citizens and uses relational composition as

a powerful operator to combine various structural entities. The design of Alloy was influenced by Z and it can be (roughly) viewed as a subset of Z.

Essential Alloy language constructs are presented below.

***Signatures*:**

A signature (`sig`) paragraph introduces a basic type and a collection of relations (called fields) in it along with types of the fields and constraints on their values. A signature may inherit fields and constraints from another signature. For example

```
sig Resource {}
```

defines a signature `Resource` with no relations associated with it.

The signature below defines a basic type `Class`, which is a subset of `Resource` defined above (`Class` extends `Resource`). Moreover, it has a field associated with it, the `instances`, that maps a class to the set of its instances, which are of the type `Resource`.

```
disj sig Class extends Resource
    {instances: set Resource}
```

The keyword `disj` preceding the definition asserts that this definition and other subsets of `Resource` are disjoint with each other.

***Functions*:**

A function (`fun`) captures behavior constraints. It is a parameterized formula that can be "applied" elsewhere. For example, in the following Alloy specification, `subClassOf` is a function that states for classes `c1` and `c2` to be of `subClassOf` relationship, the instances of `c1` must be a subset of the instances of `c2`.

```
fun subClassOf(c1, c2: Class)
    {c1.instances in c2.instances}
```

***Facts*:**

A fact (`fact`) constrains the relations and objects. A fact is a formula that takes no arguments and need not be invoked explicitly; it is always true. For example, the following fact states that `MilitaryTask` is a sub class of `MilitaryProcess`.

```
fact{subClassOf(MilitaryTask, MilitaryProcess)}
```

***Assertions*:**

An assertion (`assert`) specifies an intended property. It is a formula the correctness of which needs to be checked, assuming the facts in the model. For example:

```
assert PrepareDemolitionTaskIsMilProcess
    {subClassOf(PrepareDemolition_MilitaryTask, MilitaryProcess)}
```

Alloy Analyzer [55] is a constraint solver for Alloy that provides fully automated simulation and checking. Alloy Analyzer works as a compiler: it compiles a given problem into a (usually huge) boolean formula, which is subsequently solved by a SAT solver, and the solution is then translated back to Alloy Analyzer. Inevitably, a finite scope - a bound on the size of the domains - must be given to make the problem finite.

Alloy Analyzer determines whether there exists a model for the formula. When it finds an assertion to be false, it generates a counterexample, which makes tracing the error easier, compared to theorem provers. However, the capability of Alloy Analyzer is constrained by the way it works. Since Alloy Analyzer performs exhaustive search,

it does not scale very well. Similar to Z/EVES, Alloy specifications are in the form of *modules*, organized into a tree. Existing modules can be reused by commands `open` or `use`.

Besides Z/EVES and Alloy Analyzer, a number of Automated Theorem Provers have been implemented in the recent years [73, 82, 92] and Vampire [82] is one with very high performance. In [97], It has been chosen to make comparison with a DL reasoner FaCT++, the next-generation of the FaCT reasoner introduced above. In the comparison, core DL reasoning tasks, namely knowledge base classification and concept subsumption were considered. As the comparison turned out, Vampire is out performed by FaCT++. Based on the above result, the authors suggested that first-order reasoners, including Z/EVES and Alloy Analyzer, are best suited to be used in a hybrid way, performing some reasoning tasks DL Reasoners such as FaCT++ and RACER cannot deal with. This is exactly what we have done in our combined approach.

So far we have introduced several Semantic Web reasoning tools and software engineering proof tools. It is interesting to compare them. In Table 2.2, we summarize the strength and weakness of RACER, Z/EVES and Alloy Analyzer.

Table 2.2: Strength & weakness of the reasoning tools

Tool	**Strength**	**Weakness**
RACER	Fully automated reasoning	Kinds of reasoning tasks limited
Alloy Analyzer	Able to locate the source of the errors	Scope is limited
Z/EVES	Very expressive & powerful	Interactive proof process

2.4 Institutions & Institution Morphisms

Institutions and institution morphisms are used in this thesis to prove the correctness of the Z semantics of OWL in Chapter 5. In this section, we give a brief introduction to them. We assume the reader is familiar with the basics of category theory, including category, opposite category, functor, natural transformation, colimit, and the categories Set of sets and Cat of categories; e.g., see [59] for an introduction to this subject.

Institutions were introduced by Goguen and Burstall [31, 32] to formalize the notion of logical systems and to provide a basis for reasoning about software specifications independently of the underlying logical system chosen. The basic components of a logical system are *models* and *sentences*, related by the *satisfaction relation*. The compatibility between models and sentences is provided by *signatures*, which formalizes the notion of vocabulary from which the sentences are constructed. Modeling the signatures of a logical system as a category, we get the possibility to translate sentences and models across signature morphisms. The consistency between the satisfaction relation and this translation is given by the *satisfaction condition* which intuitively means that *the truth is invariant under the change of notation*.

Formally, an *institution* is a quadruple $\Im = (\mathsf{Sign}, \mathsf{sen}, \mathsf{Mod}, \models)$ where Sign is a category whose objects are called *signatures*, sen is a functor $\mathsf{sen} : \mathsf{Sign} \to \mathsf{Set}$ which associates with each signature Σ a set whose elements are called Σ-*sentences*, $\mathsf{Mod} : \mathsf{Sign}^{op} \to \mathsf{Cat}$ is a functor which associates with each signature Σ a category whose objects are called Σ-*models*, and $\models$ is a function which associates with each signature Σ a binary relation $\models_\Sigma \subseteq |\mathsf{Mod}(\Sigma)| \times \mathsf{sen}(\Sigma)$, called *satisfaction relation*, such that for each morphism $\phi : \Sigma \to \Sigma'$ the *satisfaction condition*

$$\mathsf{Mod}(\phi)(M') \models_\Sigma e \Leftrightarrow M' \models_{\Sigma'} \phi(e)$$

holds for each model $M' \in \mathsf{Mod}(\Sigma')$ and each sentence $e \in \mathsf{sen}(\Sigma)$. The functor sen abstracts the way the sentences are constructed from signatures (vocabularies). The functor Mod is defined over the opposite category Sign^{op} because a "translation between vocabularies" $\phi : \Sigma \to \Sigma'$ defines a forgetful functor $\mathsf{Mod}(\phi^{op}) : \mathsf{Mod}(\Sigma') \to \mathsf{Mod}(\Sigma)$ such that for each Σ'-model M', $\mathsf{Mod}(\phi^{op})(M')$ is M' viewed as a Σ-model. The satisfaction condition may be read as "M' satisfies the ϕ-translation of e iff M' viewed as a Σ-model satisfies e", i.e., the meaning of e is not changed by the translation ϕ.

We often use $\mathsf{Sign}(\Im)$, $\mathsf{sen}(\Im)$, $\mathsf{Mod}(\Im)$, $\models_{\Im}$ to denote the components of the institution $\Im$. If $\phi : \Sigma \to \Sigma'$ is a signature morphism, then the Σ-model $\mathsf{Mod}(\phi^{op})(M')$ is also denoted by $M'|_{\phi}$ and we call it *the ϕ-reduct of M'*.

The satisfaction relation is extended to sets of sentences and it is used to define the semantical consequence notion. If E is a set of Σ-sentences, then:

1. $M \models_{\Sigma} E$ if $M \models e$ for each $e \in E$.
2. $\mathsf{Mod}^{th}(\Sigma, E) = \{M \mid M \models_{\Sigma} E\}$.
3. $E \models_{\Sigma} e$ if $M \models e$ for each model $M \in \mathsf{Mod}^{th}(\Sigma, E)$. We say that e is a *semantical consequence* of E.

A specification (presentation) is a way to represent the properties of a system independent of model (= implementation). Formally, a *specification* is a pair (Σ, E), where Σ is a signature and E is a set of Σ-sentences. A (Σ, E)-*model* is a Σ-model M such that $M \models_{\Sigma} E$. We sometimes write $(\Sigma, E) \models e$ for $E \models_{\Sigma} e$.

The migration from one logical system to another is captured by institution morphism or institution comorphism. There are many variations on institution morphisms/comorphisms in the literature. We recommend [33, 94] for systematic investigations of these notions and the relations between them. Here we recall from

[33] the definition for simple theoroidal comorphism. Let $\Im = (\mathsf{Sign}, \mathsf{sen}, \mathsf{Mod}, \models)$ and $\Im' = (\mathsf{Sign}', \mathsf{sen}', \mathsf{Mod}', \models')$ be two institutions. We denote by Th the category of the specifications in $\Im$ and by Th' the category of the specifications in $\Im'$. Let $sign' : \mathsf{Th}' \to \mathsf{Sign}'$ be the forgetful functor which sends a specification (Σ', E') in Th' to its signature Σ'. A *simple theoroidal comorphism* $(\Phi, \alpha, \beta) : \Im \to \Im'$ consists of:

1. a functor $\Phi : \mathsf{Sign} \to \mathsf{Th}'$ such that there is a functor $\Phi^\diamond : \mathsf{Sign} \to \mathsf{Sign}'$ satisfying $\Phi\,;\ sign' = \Phi^\diamond$,
2. a natural transformation $\alpha : \mathsf{sen} \Rightarrow \Phi^\diamond\,;\mathsf{sen}'$, and
3. a natural transformation $\beta : \Phi^\diamond\,;\mathsf{Mod}' \Rightarrow \mathsf{Mod}$,

such that the following satisfaction condition holds:

$$M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(e) \text{ iff } \beta_\Sigma(M') \models_\Sigma e$$

for any $\Phi(\Sigma)$-model M' of $\Im'$ and Σ-sentence e of $\Im$. We extend Φ to the functor $\Phi : \mathsf{Th} \to \mathsf{Th}'$ such that if $\Phi(\Sigma) = (\Sigma', E')$, then $\Phi(\Sigma, E) = (\Sigma', E' \cup \alpha_\Sigma(E))$. In other words, $\Phi(\Sigma, E)$ is $\Phi(\Sigma)$ to which we add the sentences $\alpha_\Sigma(E)$. The functor Φ associates with each signature Σ in $\Im$ a specification $(\Sigma^\emptyset, E^\emptyset)$ in $\Im'$; this means that the definition of vocabularies in $\Im$ includes properties which are expressed in $\Im'$ by $E^\emptyset$. The natural transformation α consists of a morphism $\alpha_\Sigma : \mathsf{sen}(\Sigma) \to \mathsf{sen}'(\phi^\diamond(\Sigma))$ for each signature Σ in $\Im$; α_Σ defines the translation of Σ-sentences in $\Im$ into $\phi^\diamond(\Sigma)$-sentences in $\Im'$. The natural transformation β consists of a functor $\beta_\Sigma : \mathsf{Mod}'(\phi^\diamond(\Sigma)) \to \mathsf{Mod}(\Sigma)$ for each signature Σ in $\Im$; β_Σ says how a $\phi^\diamond(\Sigma)$-model in $\Im'$ can be seen as a Σ-model in $\Im$. The meaning of the satisfaction condition is similar to that from the definition of the institution.

Remark 1 *The definition for simple theoroidal comorphism is slightly modified from that given in [33]. If*

– *we extend* Mod'^{th} *to a functor* $\mathsf{Mod}'^{th} : \mathsf{Th}'^{op} \to \mathsf{Cat}$ *similar to* Mod' *but defined over specifications, and*

– *we denote by* mod' *the natural transformation* $\mathsf{mod}' : \Phi^{op}\,;\, \mathsf{Mod}'^{th} \Rightarrow \Phi^{\diamond op};\ \mathsf{Mod}'$ *such that for each signature* Σ $\mathsf{mod}'_\Sigma : \mathsf{Mod}'^{th}(\Phi(\Sigma)) \to \mathsf{Mod}'(\Phi^\diamond(\Sigma))$ *is the inclusion, and*

– β^{th} *is the vertical composition* $\mathsf{mod}'\,;\, \beta$*,*

then $(\Phi, \alpha, \beta^{th})$ *is a simple theoroidal comorphism as in [33].*