KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Ontology, Taxonomy and Type in Artificial Intelligence

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).
The present report, PM-krf-027, can persistently be accessed as follows:

Project Memo URL:	`www.ida.liu.se/ext/caisor/pm-archive/krf/027/`
AIP (Article Index Page):	`http://aip.name/se/Sandewall.Erik.-/2011/007/`
Date of manuscript:	2011-09-09

Related information can also be obtained through the following www sites:

KRFwebsite:	`http://www.ida.liu.se/ext/krf/`
AIP naming scheme:	`http://aip.name/info/`
The author:	`http://www.ida.liu.se/~erisa/`

Preface

This lecture note is intended for use in a graduate-level (Ph.D. students) course in Artificial Intelligence at Linköping University during 2011. It is likely that parts of it will later be repackaged as a technical report or as a chapter in a forthcoming book.

The following book is strongly recommended reading for the course, and pages 100-146 in the book are required reading:

> `Adam Pease: Ontology. A Practical Guide. Articulate Software Press, Angwin, CA 94508, USA.`

The SUMO ontology is available from the following website:

> `http://www.ontologyportal.org/`

Directly browsable files for the main part of SUMO are available at the following webpage, but one must caution that they are a few years old and therefore not entirely up-to-date:

> `http://piex.publ.kth.se/ckl-fields/sumo/page.html`

This page links to pages that present SUMO as Leonardo entityfiles, with some rudimentary structure so that each SUMO entity is presented as a Leonardo entity with attributes and properties, and with those axioms that are considered to pertain to an entity collected in a Leonardo property for it.

The previously read lecture notes in this course are assumed as known in the present lecture note. They can be found like before at

> `http://www.ida.liu.se/ext/kraic/`

The material in the lecture notes for the *Knowledge Representation Framework* and *Principles of Domain Modelling for Knowledge Representation* will be extensively used.

Chapter 1

About Ontologies

This chapter is an overview of the role of ontologies in Artificial Intelligence, and of major currently used ontology languages and ontologies. The following chapters will describe the SUMO and KRF ontologies with respect to their similarities and differences. Particular attention will be given to the representation of actions and change in the two systems, and to the interplay between the type system and the ontology in the case of KRF.

Some of the material in Chapter 1 has been obtained from Wikipedia pages and other sources on the Internet. The permission of the page owner has been obtained in each case, except for the Wikipedia pages where this is permitted according to their copyright notice.

1.1 Content and Purpose of Ontologies

The Wikipedia article on ontology in computer science [1] says:

In computer science and information science, an *ontology* is a formal representation of the knowledge by a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to describe the domain.

In theory, an ontology is a "formal, explicit specification of a shared conceptualisation." An ontology provides a shared vocabulary, which can be used to model a domain that is, the type of objects and/or concepts that exist, and their properties and relations.

Ontologies are used in artificial intelligence, the Semantic Web, systems engineering, software engineering, biomedical informatics, library science, enterprise bookmarking, and information architecture as a form of knowledge representation about the world or some part of it. The creation of domain ontologies is also fundamental to the definition and use of an enterprise architecture framework.

Contemporary ontologies share many structural similarities, regardless of the language in which they are expressed. As mentioned above, most on-

[1] http://en.wikipedia.org/wiki/Ontology_(information_science)

tologies describe individuals (instances), classes (concepts), attributes, and relations. Common components of ontologies include:

- Individuals: instances or objects (the basic or "ground level" objects)
- Classes: sets, collections, concepts, classes in programming, types of objects, or kinds of things.
- Attributes: aspects, properties, features, characteristics, or parameters that objects (and classes) can have
- Relations: ways in which classes and individuals can be related to one another
- Function terms: complex structures formed from certain relations that can be used in place of an individual term in a statement
- Restrictions: formally stated descriptions of what must be true in order for some assertion to be accepted as input
- Rules: statements in the form of an if-then (antecedent-consequent) sentence that describe the logical inferences that can be drawn from an assertion in a particular form
- Axioms: assertions (including rules) in a logical form that together comprise the overall theory that the ontology describes in its domain of application. This definition differs from that of "axioms" in generative grammar and formal logic. In those disciplines, axioms include only statements asserted as a priori knowledge. As used here, "axioms" also include the theory derived from axiomatic statements.
- Events: the changing of attributes or relations.

Ontologies are commonly encoded using ontology languages.

1.1.1 Formal vs Lightweight Ontologies

Citation from [2]

A variety of ontologies form a continuum from lightweight, rather informal, to heavyweight, and formal ontologies. The lightweight ontology approach and the formal ontology approach are often used differently and have different strengths and weaknesses. Lightweight ontologies usually are taxonomies, which consist of a set of concepts (i.e., terms, or atomic types) and hierarchical relationships among the concepts. It is relatively easy to construct a lightweight ontology. To use a lightweight ontology for interoperability purposes, all parties need to agree on the exact meaning of the concepts. Reaching such agreements can be difficult. The lightweight ontology and the agreements together form a standard that all parties uniformly adopt and implement. That is, a lightweight ontology is often used to support strict data standardization. In contrast, a formal ontology uses axioms to explicitly represent subtleties and has inference capabilities. It can support data standardization in a different way, that is, the agreements are explicitly specified in the ontology. More often, a formal ontology is used to allow for data heterogeneity and to support interoperability, in which case

[2] http://web.mit.edu/smadnick/www/wp/2006-06.pdf

the different interpretations and representations of data are explicitly captured in the ontology. In either case, a formal ontology disambiguates all concepts involved. Once created, a formal ontology can be relatively easy to use. But it often takes tremendous effort to create a formal ontology due to the level of detail and complexity required.

To summarize, lightweight ontologies are often used as data standards; as artifacts, they are simple, and thus easy to create, but difficult to use. Formal ontologies are often used to support interoperability of heterogeneous data sources and receivers; as artifacts, they are complex and difficult to create, but easy to use. Either approach has its weakness that limits its effectiveness.

(The Wikipedia article on lightweight ontologies is not recommended).

Domain ontologies and upper ontologies

The following distinction is mostly applicable for formal ontologies; lightweight ontologies typically do not use it or need it.

A domain ontology (or domain-specific ontology) models a specific domain, or part of the world. It represents the particular meanings of terms as they apply to that domain. For example the word card has many different meanings. An ontology about the domain of poker would model the "playing card" meaning of the word, while an ontology about the domain of computer hardware would model the "punched card" and "video card" meanings.

An upper ontology (or foundation ontology) is a model of the common objects that are generally applicable across a wide range of domain ontologies. It contains a core glossary in whose terms objects in a set of domains can be described. There are several standardized upper ontologies available for use, including Dublin Core, GFO, OpenCyc/ResearchCyc, SUMO, and DOLCE. WordNet, while considered an upper ontology by some, is not strictly an ontology. However, it has been employed as a linguistic tool for learning domain ontologies.

The Gellish ontology is an example of a combination of an upper and a domain ontology.

Since domain ontologies represent concepts in very specific and often eclectic ways, they are often incompatible. As systems that rely on domain ontologies expand, they often need to merge domain ontologies into a more general representation. This presents a challenge to the ontology designer. Different ontologies in the same domain can also arise due to different perceptions of the domain based on cultural background, education, ideology, or because a different representation language was chosen.

1.2 Ontology Languages

Most of the representation languages that were described in the lecture note about principles of domain modelling are used for representing ontologies, in particular KIF, CycL and OWL. The following are some additional

languages that have been developed specifically for representing ontologies (quotation from the Wikipedia article on ontologies [3])

- The *Common Algebraic Specification Language* is a general logic-based specification language developed within the IFIP (International Federation of Information Processing) working group 1.3 "Foundations of System Specifications" and functions as a de facto standard in the area of software specifications. It is now being applied to ontology specifications in order to provide modularity and structuring mechanisms.
- *Common logic* is ISO standard 24707, a specification for a family of ontology languages that can be accurately translated into each other.
- *DOGMA* (Developing Ontology-Grounded Methods and Applications) - additional information not available.
- The *Gellish language* includes rules for its own extension and thus integrates an ontology with an ontology language.
- The *Integrated Definition for Ontology Description Capture Method* (IDEF5) is a software engineering method to develop and maintain usable, accurate, domain ontologies.
- The *Rule Interchange Format* (RIF) and *F-Logic* combine ontologies and rules.
- The *Semantic Application Design Language* (SADL) captures a subset of the expressiveness of OWL, using an English-like language entered via an Eclipse Plug-in.
- OBO, a language used for biological and biomedical ontologies.

1.2.1 The RIF Language

Wikipedia says:
RIF is part of the infrastructure for the semantic web, along with (principally) RDF and OWL. Although originally envisioned by many as a "rules layer" for the semantic web, in reality the design of RIF is based on the observation that there are many "rules languages" in existence, and what is needed is to exchange rules between them. *End of citation.*

In practice RIF is therefore a family of languages, rather than a single language, and in terms of the present compendium one may consider RIF as a language style and its various so-called dialects as languages using that style. The following are simple examples from some of these 'dialects'.

The Production Rules Dialect

The Production Rules Dialect (PRD) can be used to model production rules. Notably features in PRD include negation and retraction of facts (thus, PRD is not monotonic). PRD rules are order dependent, hence conflict resolution strategies are needed when multiple rules can be fired. The PRD

[3] http://en.wikipedia.org/wiki/Ontology_(information_science)

specification defines one such resolution strategy based on forward-chaining reasoning.

```
Prefix(ex <http://example.com/2008/prd1#>)
(* ex:rule_1 *)
Forall ?customer ?purchasesYTD (
 If   And( ?customer#ex:Customer
           ?customer[ex:purchasesYTD->?purchasesYTD]
           External(pred:numeric-greater-than(?purchasesYTD 5000)) )
 Then Do( Modify(?customer[ex:status->"Gold"]) ) )
```

The Uncertainty Rule Dialect

The Uncertainty Rule Dialect (URD) supports a direct representation of uncertain knowledge, as in the following example.

```
Document(
  Import (<http://example.org/fuzzy/membershipfunction >)
  Group
  (
    Forall ?x ?y(
        cheapFlight(?x ?y) :- affordableFlight(?x ?y)
    )  / 0.4
    Forall ?x ?y(affordableFlight(?x ?y))  / left_shoulder0k4k1k3k(?y)
  )    )
```

It is debatable whether these RIF languages should best be described as knowledge representation languages, specialized ontology languages or high-level programming languages.

1.2.2 Common Logic

Wikipedia writes:
Common logic (CL) is a framework for a family of logic languages, based on first-order logic, intended to facilitate the exchange and transmission of knowledge in computer-based systems.

The CL definition permits and encourages the development of a variety of different syntactic forms, called "dialects." A dialect may use any desired syntax, but it must be possible to demonstrate precisely how the concrete syntax of a dialect conforms to the abstract CL semantics, which are based on a model theoretic interpretation. Each dialect may be then treated as a formal language. Once syntactic conformance is established, a dialect gets the CL semantics for free, as they are specified relative to the abstract syntax only, and hence are inherited by any conformant dialect. In addition, all CL dialects are equivalent (i.e., can be mechanically translated to each other), although some may be more expressive than others.

The standard includes specifications for three dialects, the Common Logic Interchange Format (CLIF), the Conceptual Graph Interchange Format (CGIF), and an XML-based notation for Common Logic (XCL). The semantics of these dialects are defined by their translation to the abstract syntax and semantics of Common Logic. Many other logic-based languages could also be defined as subsets of CL by means of similar translations;

among them are the RDF and OWL languages, which have been defined by the W3C (World-Wide Web Consortium). *End of citation.*

The following is a simple example of these formats:

```
CLIF:  (exists (x y) (and (Red x) (not (Ball x)) (On x y)
                          (not (and (Table y) (not (Blue y)))) ))
CGIF:  ~[[*x] [*y] (Red ?x) ~[(Ball ?x)] (On ?x ?y)
                  ~[(Table ?y) ~[(Blue ?y)]]]
```

This shows how CLIF uses S-expression style and CGIF uses its own special format for representing logic formulas using the standard (Latin-1) computer character set. The peculiar conventions of CGIF include the use of square brackets for enclosing a conjunction (and-expression) and a universal quantifier, and the use of the asterisk for marking the quantification of a variable.

Notice that the concept of dialect in Common Logic is almost opposite of dialects in RIF. RIF dialects are languages for different purposes that use a more or less common style, whereas the point with Common Logic dialects is to allow different styles for the same, or at least overlapping semantic content.

1.2.3 The Gellish Language

The Wikipedia article about Gellish [[4]*] writes:*
Gellish is a controlled natural language, also called a formal language, in which information and knowledge can be expressed in such a way that it is computer-interpretable, as well as system-independent. Gellish is a structured subset of natural language that is suitable for information modelling and knowledge representation and as a successor of electronic data interchange. From a data modeling perspective, it is a generic conceptual data model that also includes domain-specific knowledge and semantics. Therefore, it can also be called a semantic data model. The accompanying Gellish modelling method thus belongs to the family of semantic modelling methods.

The data model in Gellish is based on binary relations between entities, similar to the model in OWL.

Etymologically speaking, "Gellish" is originally derived from "Generic Engineering Language." However, it is further developed into a language that is also applicable outside the engineering discipline.

1.2.4 The SADL Language

A simple example of the use of SADL, from its webpage:

```
shapes-top.sadl
   uri "http://ctp.geae.ge.com/iws/shapes_top".

   Shape is a top-level class.
```

[4] http://en.wikipedia.org/wiki/Gellish

```
   area describes Shape has values of type float.

shapes-specific.sadl
   uri "http://ctp.geae.ge.com/iws/shapes_specific".

   import "file://shapes-top.sadl" as shapes-top.

   Circle is a type of Shape.
   radius describes Circle has values of type float.

   Rectangle is a type of Shape.
   height describes Rectangle has values of type float.
   width describes Rectangle has values of type float.
```

Reasoning over a set of SADL documents takes two basic forms. *Validation* of a model involves checking the model for contradictions or inconsistencies. *Rule processing* involves examining the rules in the model in light of the current instance data to see if any of the rules can "fire" to infer additional information. Two reasoners are integrated with the SADL Integrated Development Environment (SADL-IDE).

A systematic transformation from SADL to OWL has been defined.

1.2.5 The IDEF5 Method and Language

The digit '5' in the acronym IDEF5 is not a version generation number, but represents the fact that there is an IDEF family of modelling languages that serve different and complementary purposes, and IDEF5 is the particular language used for ontologies in this family. This family of languages was developed by the U.S. Air Force in the early 1990's, and is presently maintained and used by a commercial company, Knowledge Based Systems, Inc.

The IDEF5 method has three main components: A graphical language to support conceptual ontology analysis, a structured text language for detailed ontology characterization, and a systematic procedure that provides guidelines for effective ontology capture. The graphical language appears to be the primary representation.

1.3 Published Formal Ontologies

A substantial number of proposed formal ontologies have been published, both general-purpose ones and specialized ontologies for different disciplines or areas of knowledge or application. The following are some of the more important general-purpose ontologies.

- The Cyc ontology, [5]
- The Suggested Upper Merged Ontology, SUMO, [6]
- The Generalized Upper Model, GUM

[5] http://en.wikipedia.org/wiki/Cyc
[6] http://en.wikipedia.org/wiki/Suggested_Upper_Merged_Ontology

- The Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE), [7]

Ontologies such as these are fairly elaborate things, often beginning with an almost philosophical discussion of types of concepts and their relationships and uses. It is not the purpose of the present compendium to address those issues, and we shall merely make a few notes about the technical and administrative aspects of some of the ontologies, and in particular how they relate to the styles and representation languages that have been described in earlier chapters.

1.3.1 The SUMO Ontology

SUMO – the Suggested Upper Merged Ontology – is an open resource that uses the KIF representation language, which is essentially first-order logic. It will be discussed further in the next chapter.

1.3.2 The DOLCE Ontology

For a description of the DOLCE ontology, please refer to the above-mentioned webpage and to the section about DOLCE i the SUMO book.

The *WonderWeb Foundational Ontologies Library* (WFOL) is intended to contain a variety of ontology documents together with tools for relating them. DOLCE is the first item in that library.

1.3.3 The Protégé Ontology Library

The Protégé Ontology Library [8] contains a considerable number of ontologies, in particular in the two formats that are supported by Protégé, i.e. OWL and OKBC. Contents range from the very general, such as DOLCE, to the quite specific e.g. `Daycare` - "A demo ontology about a childcare center showing the use of SWRL for reasoning," or `Camera` - "An OWL ontology about the individual parts of a photo camera."

1.3.4 The Cyc Ontology

The Cyc ontology uses the CycL representation language (see the separate lecture note about knowledge representation languages). It is largely proprietary, by contrast to all the preceding ontologies that are open-source and can be used freely.

[7] http://www.loa-cnr.it/Papers/DOLCE2.1-FOL.pdf
[8] http://protegewiki.stanford.edu/wiki/Protege_Ontology_Library

Chapter 2

Taxonomies in SUMO and KRF

As described in Chapter 1, ontologies are normally organized using a hierarchy of concepts together with informal and formal characterizations of these concepts. The formal characterization uses restrictions, rules and axioms. Usually the concept structure is strictly hierarchical, so that each node has only one superior node. This hierarchical structure is called a *taxonomy.*

The present chapter will discuss the taxonomies used by SUMO (Suggested Upper Merged Ontology) and the KRF (Knowledge Representation Framework) in a comparative way. The intention is that this will be helpful for understanding issues in the design of such taxonomies.

2.1 Background

2.1.1 About SUMO

The acronym SUMO stands for "Suggested Upper Merged Ontology" is presented as follows on the SUMO webpage: *The Suggested Upper Merged Ontology (SUMO) and its domain ontologies form the largest formal public ontology in existence today.* In practice the acronym SUMO is used both in the restricted sense for the top-level or "upper" structure, and for the entire system that also includes the domain ontologies.

The upper ontology defines a considerable number of *entities* and contains *axioms* that are intended to define the meaning of these entities. The files for the upper ontology also contain verbal descriptions for entities which are intended as guidance for the human reader who wishes to understand the structure. Axioms include both ground literals that express relationships between entities, and general axioms using quantifiers that range over the domain of entities. A *taxonomy* over the entities is expressed using ground literals.

Some of the domain ontologies consist entirely of literals that express elementary facts. This applies for example for the domain ontologies for `people` and for `countries and regions`. Other domain ontologies extend

the upper ontology in more substantial ways by also including non-ground axioms.

The SUMO ontology is large; it contains around 20.000 terms and 70.000 axioms when all the domain ontologies are included.

2.1.2 About Ontology in KRF

The work on the Knowledge Representation Framework, the Leonardo system and the Common Knowledge Library (CKL) needs to use a formal ontology, or at least some aspects of a formal ontology, for the following purposes:

- In order to have a coherent structure and a systematic terminology for entity types that are used for self-description purposes in the Leonardo system, where one of the design goals is that its software shall contain an exhaustive description of itself
- In order to have a coherent and systematic structure for the types that are used in the Common Knowledge Library
- In order to have an ontology that is consistent with the material about knowledge representation in the lecture notes for the present course. This applies in particular to the logic for reasoning about actions and change, both for the general predicates and functions that are used there, and for the required extensions when specific action verbs are introduced
- The logical treatment of defeasible multiple inheritance similarly needs to be related to the taxonomical aspect of ontologies.

The SUMO ontology and other existing ontologies were not considered entirely adequate for these purposes, and our approach is therefore to introduce our own structure, while at the same time keeping track of how it relates in particular to SUMO, and being open to re-use small or large sections of SUMO whenever appropriate. This working structure will be referred to here as the *KRF taxonomy* or the *KRF ontology.* We emphasize that at this point it is working material and by no means a finished ontology. This approach is deemed reasonable for use within our own project. For external presentation it will be a later question whether a finished KRF ontology shall be presented, or whether this will result in a proposed extension of SUMO.

In comparing the approaches it should be noticed that for those SUMO domain ontologies that consist of ground clauses, the natural counterpart in the KRF approach is the existing Common Knowledge Library which consists of more than 60.000 entities, each with a nontrivial number of attributes.

We proceed now to the discussion of the top level of the taxonomies in SUMO and KRF.

2.2 Top Levels and Correspondences

We shall represent taxonomy structures using indentation, as in the following diagram for the topmost level of the SUMO taxonomy.

```
Entity
   PhysicalEntity
      Process
      Object
         Self-connectedObject
         Agent
         Collection
         Region
   AbstractEntity
      Attribute
      Proposition
      Graph
      GraphElement
      Set
      Quantity
         Number
         PhysicalQuantity
```

The taxonomy continues with additional subtypes for the types shown here, but this should give the general idea. The following is, by comparison, a part of the topmost level of the KRF taxonomy.

```
item
   entity
      thing
         process
         individual-entity
            tangible-entity
            intangible-entity
            social-entity
         compound
      spatial-entity
      temporal-entity
      quality
   descriptor
      attribute
      dynamic-descriptor
```

SUMO entities are customarily written with a combination of capital and small letters in the style that is suggested by these examples, whereas KRF entities and constructs are written with small letters only, and using hyphens for forming multi-word symbols. The use of these conventions here will make it easier to keep track of which symbol belongs to which structure. [1]

In several cases there is a more or less exact correspondence between nodes in the two structures. In particular, `Process` in SUMO corresponds to `process` in KRF; these refer to processes in the same sense as in our lecture notes, so that a process is one specific instance of an action.

[1] A few entities in the KRF ontology are actually written with capital letters, but we shall write them with small letters here to avoid confusion.

Likewise, `Region` in SUMO corresponds to `spatial-entity` in KRF: these are used e.g. for geographical areas such as continents, islands, and countries. There is an allowance to also use these for things that have both a geographical extent and other characteristic properties, as is the case for countries.

A `Self-connectedObject` in SUMO has two subtypes namely `Substance` and `CorpuscularObject.` The amount of water in a glass is an example of a `Substance` whereas a person, a car, and a bee may be examples of `CorpuscularObject.` This substructure corresponds to `tangible-entity` with its two subtypes `amorphous-entity` and `morphous-entity.`

A `Collection` in SUMO corresponds to a combination of two types in KRF, namely `social-entity` and `compound.` A `social-entity` is for example a family, an enterprise, or the collection of guests at a dinner. A `compound` is for example the set of subscribers to a computer newsletter.

The type `PhysicalQuantity` has a subtype that matches `temporal-entity.` The type `Attribute` partly patches `quality.` The types for `Agent` and for `intangible-entity` are less easy to match.

These examples have shown that in some cases the difference between the two structures is mostly a question of naming, and they should also have given some idea as to what may be included under the various types in a hierarchy. We proceed now to some of the differences. This will actually be more interesting since it will illustrate a number of taxonomy design issues.

2.3 Difficult Matches

In general one can say that for the `entity` or world-oriented part of the KRF taxonomy it is easy to identify the correspondences with the SUMO taxonomy, with one noteworthy exception on each side. In SUMO there is the type called `Proposition` which has a quite broad meaning. It does not merely refer to a proposition in the sense of formal logic, but it can be applied to anything that makes a statement about things, including for example a set of axioms, an article, or a book.

A partial counterpart for this can be found in the KRF taxonomy under `intangible-entity` which was included in the figure above and which can be expanded as follows:

```
entity
   thing
       process
       individual-entity
          tangible-entity
          intangible-entity
             computation-entity
             cognition-entity
             communication-entity
             convention-entity
             info-entity
             model-entity
             abstract-entity
          social-entity
```

`compound`

The type `Proposition` corresponds partly to `info-entity` here. In this structure, a `computation-entity` may be an command verb in an OS shell language, for example, a `cognition-entity` may be a method for doing a particular task, a `communication-entity` may be an email message, and a `convention-entity` may be a natural language or a grammar rule in such a language.

The presence of this relatively rich structure at a high level in the KRF taxonomy may be attributed to the fact that the taxonomy has been developed together with the Leonardo software system that implements the KRF, and one requirement on the design was that the software system should characterize itself, and in fact all aspects of itself, using its own representation system. The SUMO taxonomy did not have such a requirement, and as a consequence types corresponding to these ones must be found further down in the hierarchy.

The situation is different with respect to other parts of the KRF ontology since they are related to an underlying *type system* that is oriented towards the needs of the elementary knowledgebase, in the sense of entities and their attribute values as represented in entityfiles, and to some extent also to the needs of the software development. The type system is defined first, the main purpose of the taxonomy is to characterize phenomena in the world which is its `entity` part, and the reason for having a `construct` part at all is in order to provide a bridge between the taxonomy proper and the type system.

In the SUMO ontology, entities are characterized using a combination of an English-language explanation and a number of axioms that are written in the KIF language, which is essentially first-order logic. Therefore it is a natural choice (although not a necessary one) that additional facts that originate from an application area will also be expressed in KIF, so that the fact base and the taxonomy can be combined smoothly.

In the case of KRF it is explicitly assumed that facts from the application area will be expressed using the Common Expression Language (CEL), which means that they besides first-order logic they can also use constructs such as sets, sequences and records. This is one of the reasons why a non-trivial type system is needed. It is furthermore assumed that applications will use the design principles described in [2] in particular for the representation of actions, processes, and changes over time. The corresponding representational background for SUMO has not been documented to the same extent.

2.4 Actions and Change in KRF Ontology

We shall now proceed to look more closely at the representation of actions and change according to the two ontologies. In the case of KRF it is based on the explicit-time logic of actions and change that has been described in the lecture note on principles of domain modelling (as listed in the Preface). This is therefore one example of how an ontology is not necessarily defined in isolation and out of philosophical or intuitive considerations; it may also

[2] Principles of Domain Modelling for Knowledge Representation

be designed so as to go well with a formal-logical view of certain kinds of information.

Quite briefly, the KRF representation for actions and change is organized around two predicates `H` (for Holds) and `D` (for Do). In a proposition of the form

```
[H t f v]
```

the first argument is a timepoint, the second argument is a feature, and the third argument is a value for that feature. For example,

```
[H year.2011 (the: has-president of USA) Obama.Barack]
```

uses a feature formed by the function `the` with two arguments, an attribute `has-president` and an entity `USA.` (The `of` that appears in an argument position is only for cosmetic purposes). The `H` predicate assigns a value to that feature at the time in question. This representation makes it possible to assign different values to the feature at different points in time. It also makes it possible to use a richer "time" concept than merely a linear time axis; one may introduce timepoints in hypothetical futures for example.

Features may be viewed as a reification of an underlying binary relation, so that the timeless proposition

```
[has-president USA Obama.Barack]
```

is converted to the expression shown above if one wishes to state that it applies during the year 2011, for example.

The `D` predicate is used for propositions of the form

```
[D s t a]
```

where `s` and `t` are timepoints and `a` is an action, for example

```
[D 2011-04-06 2011-04-08 [visit :by John :at Stockholm]]
```

The third argument of the `D` predicate is syntactically a *record* and not a proposition, so the representation is correct for first-order logic. Records are composite terms in the same sense as sets and sequences are.

An action may be *reified* whereby an entity is introduced for one particular occurrence of the action, i.e. for a process.

Features are entities so they have already been reified.

This representation is essentially the same as is used in time and action logic and the modern event calculus, and the situation calculus can be obtained from it by assuming forward-branching time, adding a situation successor function, and adding one axiom. Please refer to the lecture notes for reasoning about actions and planning for further details about these representational alternatives.

The following is how this representational background is used in the KRF ontology. Selected parts of the top level of the taxonomy are expanded as follows.

```
item
   entity
   |  thing
   |     process
```

```
|     individual-entity
|     compound
|  quality
|  spatial-entity
|  temporal-entity
descriptor
|  attribute
|  dynamic-descriptor
|     feature
|     relationship
|        action
|        characterization
|  type-descriptor
|     expression-type
|     formant
|        (formant: quality)
|        (formant: attribute)
|        (formant: dynamic-descriptor)
|           (formant: feature)
|           (formant: relationship)
|              (formant: action)
|  quantity
|     number+sort
```

Features and actions are considered as instances of the `descriptor` type, and in particular as instances of their subtypes `feature` and `action,` respectively. The function `the:` is therefore a function that maps an `attribute` and a `thing,` to a `feature.` (Notice that qualities can not have time-dependent properties). Since all entities must have a type and be accounted for in the ontology, there is also the type of `formant` with its various subtypes, and the entity `the:` is an instance of the type `(formant: feature).` The function `formant:` has a type as argument, and a subtype of `formant` as its value.

Similarly for actions, each action verb is considered as a member of the type `(formant: action)` and the record expressing an action is an instance of the type `action.`

There is no function for creating reified actions, but when reified actions are created in the way described in Section 1.8 of "Principles of Domain Modelling" then they obtain the type `process.` (Please recall that instances of actions are called processes in the KRF terminology).

If one should wish to assign attributes to an action as such, rather than to specific instances of it, then one would need a function that maps an action to its generic reification. No such function has been defined at this point, since there has not been any need for it, but it is of course possible to use predicates that have actions as arguments - arguments of actions do not need to be entities.

2.5 Actions and Change in SUMO Ontology

The SUMO ontology uses three major representations for actions and change; they will be described here using CKL notation for the sake of uniformity.

(In these examples, which do not use quantifiers or other complications, the difference is actually only between the use of square brackets or round parentheses).

The assignment of attribute values that vary over time uses a modal operator `holdsDuring` and the same binary predicate `attribute` as for assignments that do not change over time. The predicate `attribute` of two arguments is in general used like in the following literal:

```
[attribute car-14 has-sunroof]
```

Such a statement expresses that `car-14` has the attribute `has-sunroof` regardless of time. The following proposition is an example where the assignment holds during a time interval:

```
[holdsDuring Today [attribute car-14 green]]
```

Notice by the way that such an attribute assignment in SUMO representation consists of only two parts, whereas in KRF representation there is an entity, an attribute, and a value, such as in

```
[H today (the: has-color of car-14) green]
```

On the other hand there is of course nothing to prevent one from introducing a particular predicate such as `has-color` in the SUMO representation, so as to be able to write

```
[has-color car-14 green]
[holdsDuring Today [has-color car-14 green]]
```

Notice also that the SUMO type `Attribute` is used for entities that are intended for use as attribute values, such as `has-sunroof,` whereas the KRF type `attribute` is used for the designator of the attribute, such as `has-color` in the KRF example. (The KRF type called `category` is analogous to the SUMO `Attribute` but it is only used for pragmatic purposes in the software).

A partly related issue concerns quantitative descriptors that vary over time, such as the selling prize of a car or the temperature of the water in a bucket. The SUMO taxonomy subdivides `PhysicalQuantity` into `ConstantQuantity` and `FunctionQuantity` as follows:

```
Entity
   PhysicalEntity
      Process
      Object
   AbstractEntity
      Attribute
       ...
      Quantity
         Number
         PhysicalQuantity
            ConstantQuantity
            FunctionQuantity
```

A `ConstantQuantity` consists of a number and a sort, such as for example "4 meters" whereas a `FunctionQuantity` is a function from a set of type `ConstantQuantity` to another such set, where presumably all members of the argument domain must have the same sort, and similarly for the value

domain. The argument domain can be an interval on the timeline, for example.

The KRF representation represents situations such as these using the feature construct and the `H` predicate.

A question that arises in this context is whether a `FunctionQuantity` *is* only a mapping, or whether it also includes the characterization of the entity having the quantity in question. For example, suppose you have two buckets of water that stand side by side, so that their temperature as a function of time is exactly the same during the time interval in question. Do you then have one single `FunctionQuantity` or two different ones? (In the case of the KRF representation, two features `(the: a of e)` and `(the: a' of e')` are only equal if `a` equals `a'` and `b` equals `b'` so each bucket of water has its own temperature feature).

For processes, finally, the SUMO representation uses reified action expressions to represent specific processes, and it also defines a reportoire of "case roles" for use in these. For example, the sentence "Lars travelled to Uppsala on April 6, 2011" would be represented as the conjunction of the following propositions, for some suitable choice of the first argument in the literals:

```
[instance travel-43 travel]
[agent travel-43 lars]
[destination travel-43 uppsala]
[time travel-43 2011-04-06]
```

This may be compared with the primary representation in KRF which is

```
[W 2011-04-16 [travel :by lars :to uppsala]]
```

while it is also foreseen that such a proposition can be reified, obtaining the same representation as is used by SUMO. The primary representation in KRF is more compact and is sufficient for use in action effect laws, but the less compact, reified representation is needed if one wishes to add further information about the process in question.

SUMO also introduces an ontology for processes in the form of abstract verbs and relates them to the case roles. This is an important issue that deserves separate study.

This comparison between the SUMO and KRF representations for actions and change has shown that there are significant differences between the two systems, but they are intertranslatable to quite a large extent. One may therefore ask what are the situations (if any) where the difference is of any importance, how strong are the preferences in those particular situations, and how big are the disadvantages of having to deal with alternative representations?

With respect to what are the situations where a particular choice makes a difference, the following can be said from the KRF point of view.

- The KRF representation using features and the `H` predicate stays strictly within first-order logic, whereas the `holdsDuring` predicate in SUMO is a modal operator. There is a general argument in favor of staying within first-order logic for computational reasons. On balance, if the use of the `holdsDuring` operator is restricted to having ground literals as its second argument, then the representation can

easily be transformed to a first-order one. If there is no such restriction, so that expressions using propositional connectives and quantifiers can be used as the second argument, then this representation is significantly more expressive, but also significantly more difficult to process.

- The use of features and the `H` predicate is essential for solutions to the frame problem when reasoning about actions, in particular since features are also used as an argument to the occlusion predicate.
- The use of action expressions in the KRF representation makes it possible to write effect rules for actions much more compactly than if only the reified representation is available, but it does not provide any additional expressivity for effect rules.
- There are also examples where one wishes to make a statement about an action in general, and not merely about one instance, or all instances of that action. It may be argued, for example, that the sentence "eating animals is wrong" is not correctly expressed by "every activity where someone eats an animal, is wrong," since one may wish to allow for exceptions, and also since value statements may be applied even to action expressions for which there are no instances at all. This speaks in favor of the use of KRF action expressions as a complement to the reified representation.

With respect to the last question, the problems can only appear when one tries to combine actual software that uses different representation. As long as the exchange between representations only occurs on the level of publication, it is not much of an issue, provided of course that each representation is clearly defined, and that the translation between them has been defined.

In his book, Adam Pease discusses the differences between SUMO and several other representations and ontologies, and the differences that he identifies are often more profound than what we have seen between SUMO and KRF, both because of a lack of effective expressiveness in other approaches, and (sometimes) because of lack of precision in the definitions.

Although the SUMO ontology has many strong sides, it is not necessarily the last word on ontology, and some other researchers may also have bigger differences with SUMO than what I have. In fact there is a tension between two objectives with respect to ontologies: one one hand the idea with an ontology is that it should be a common representation that facilitates cooperation, but at the same time we know that further development is needed so it would be premature to concentrate all activity in our field to a single one. Doing so would be an obstacle to continued development. The use of a healthy variety of alternative representation systems, not too many and not too few, is in the interest of research.

Representation systems and ontologies are large and complex designs that embody a large number of design decisions. The discussion in this section has therefore only scratched the surface of its topic, but it is offered since it provides some examples of the issues that may arise.

Chapter 3

Type Structure and Ontology

The KRF ontology is complemented by a *type structure* which will be introduced in this chapter. We shall describe the motivation for the type structure and how it relates to the taxonomy. The next chapter will provide technical details for one part of the type structure.

3.1 Type and Subsumption

Consider the elementary situation where "car 14 is green." It is fairly clear that we will like to say that car 14 is an *instance* of the type of automobiles, and green is an instance of the type of colors. In order to make this structure explicit, we should also consider `automobile` and `color` as entities, and in particular they will have a place in our ontological taxonomy, where they are subsumed by `vehicle` and `visual-quality,` respectively.

It is quite possible to stop there, but in the case of the KRF ontology we wish to make it a consistent principle that every entity shall have a type. Besides the aesthetic aspect, types are used in order to charaterize what attributes a particular entity can have, what is the permissible structure of those attributes, what are is the permissible structure for arguments and values of functions, and what is the permissible structure for the case roles of an action.

Therefore we need to have an answer to the question as to what is the type of the entities such as `vehicle, visual-quality, automobile,` and `color.` In the KRF ontology this is handled by making a clear distinction between *type membership* and *subsumption,* with two separate inclusion relations.

With respect to type membership, there are essentially three *type levels.* Entities such as `vehicle` and `has-color` belong to level 2; entities such as car 14 and `green` belong to level 3. Each entity of level 3 shall have a type that is an entity of level 2; each entity of level 2 shall have a type that is an entity of level 1. The type of an entity of level 1 is also on level 1, so circularities are allowed on that level.

Subsumption is widely applicable among entities on level 2, as our example has illustrated, and in fact the KRF taxonomy is a structure for that level only. Two of the main branches in the KRF taxonomy are `thing` and `quality`. An interesting difference between them is that *for level-3 entities whose type is* `quality` (including a type that is subsumed by `quality`) it is also often natural to use the subsumption relation, whereas the same rarely holds for entities whose type is (subsumed by) `thing`. For example, `red` is subsumed by `brownish-red` and `pale-read,` but there is no reasonable way of subsuming a particular car. On the other hand, the part-of relation is very often applicable for entities that are instances of concepts in the `thing` branch, while this is not the case in the `quality` branch.

The number of entities in level 1 is quite small. It includes `thingtype` which is the type for the level-2 entity `thing` and for all level-2 entities that are subsumed by it. Similarly there is `qualitytype` which is the type for all items subsumed by the level-2 entity `quality` including that one itself. There is `spatial-entity-type` and `temporal-entity-type` corresponding to `spatial-entity` and `temporal-entity` which are the top levels for their respective branches on level 2. Finally there are two most general entities `type` and `supertype` that are above everything else on level 1, and which is where the circularity takes place.

The KRF ontology is mostly concerned with level 2 in this structure. Items on level 3 are represented in specific applications and in the *Common Knowledge Library* (CKL) which is a relatively large knowledgebase consisting of elementary facts about individual level-3 entities.

3.2 Data Items and Ontology Items

Besides the considerations of type level, there is an issue about how entities in the ontological taxonomy relate to the data structures of the software system using the ontology. This is particularly important in the KRF ontology since the structure of the software system is an integrated part of the Knowledge Representation Framework, and since this framework is implemented as the Leonardo software system.

The approach to this problem uses a taxonomic structure that has the top levels shown in the following diagram:

```
item
   data-item
      scalar
         string
         number
      data-entity
         symbol
         composite-entity
      set
      sequence
      record
      term
   descriptor
      attribute
      dynamic-descriptor
```

```
      feature
      relationship
         action
         characterization
   type-descriptor
      expression-type
      formant
         (formant: quality)
         (formant: attribute)
         (formant: dynamic-descriptor)
             ...
   quantity
      number+sort
entity
   thing
   quality
   spatial-entity
   temporal-entity
```

The three branches of this structure are headed by `data-item, descriptor,` and `entity,` respectively. The first of these lists the syntactic types for Knowledge Representation Expressions (KRE) in an obvious way, except that what is called `entity` in the lecture note on the Knowledge Representation Framework is called a `data-entity` here in order to avoid ambiguity. (There is also an issue about whether `variable` and `tag` should also be included, but this is marginal for the present discussion). The branch for `entity` contains things that are of obvious concern for the ontology since they refer directly to phenomena in the real world. The branch for `descriptor` is in an intermediate position.

All instances of the types in the `descriptor` and `entity` branches are *represented as* instances of some type in the `data-item` branch. Usually they are instances of `data-entity` but sometimes they are instances of `record,` for example for instances of `relationship` and of `number+sort.`

Instances from the `descriptor` branch are often composite entities, for example, entities formed using the function `the:` The arguments for these functions may come from any of the three branches. Therefore, for example, if one should wish to introduce a function that maps an integer such as 1789 to the concept of year 1789, then that function would have a natural place as an instance of a type in the `descriptor` branch, and more particularly as an instance of `(formant: year-entity)` for example. A function that takes a number and an entity representing a measurement sort, such as `kilogram,` and maps this to a `record` that represents an instance of `number+sort` will accordingly be an instance of the type `(formant: number+sort).`

We consider it convenient to let the ontology in the proper sense and its taxonomy focus on the `entity` branch of the above structure, since the items in the `data-item` and `descriptor` branches are fairly mechanical and formal. In particular their relation to the modelling of phenomena in the real world is merely to provide some notational instruments for this, rather than to participate in the modelling as such.

3.3 Data Items and Descriptor in SUMO

The SUMO ontology differs from the approach of the KRF ontology by integrating the counterparts of data items and descriptors into the main ontology. This means that concepts such as `set` and `sequence` are the subject of characterization in the ontology, including the introduction of axioms that characterize the elementary properties of such constructs.

Some correspondences are easy to see: `Number` matches `number`, `ConstantQuantity` matches `number+sort`, and `Set` matches `set`, at least to the extent that one only considers sets that are defined by enumerating their members.

(It is intended to add more details about the SUMO approach in this respect here, together with a discussion of the pros and cons of these approaches).

3.4 Type Descriptors for Composite Expressions

Returning to the KRF perspective, suppose you wish to characterize an attribute (in the KRF sense) where the attribute value shall be either an entity whose type is `automobile` or a subtype thereof, or a set of such entities. This will be an example of a *type descriptor for a composite expression.* It may be of interest when specifying how the attribute in question may be used, or for a routine that checks compliance for a large collection of attribute-value assignments. Such a type descriptor may also be needed, for example, for characterizing permitted arguments and possible values for a function with a procedural definition.

However, the types that occur in the ontological taxonomy, and in its level 2 in particular, tend to be atomic symbols that are not well suited for characterizing structured data items. The KRF type system therefore contains a number of operators that can be used for forming descriptions such as these, for example

```
(join automobile (setof automobile))
```

for the aforementioned example. These operators and expressions must then in turn have a place in the type system, and so on until closure is obtained. All of this is also included in the `descriptor` branch of the entire system for types and taxonomy. This means that the `descriptor` branch contains facilities for describing both entities, data items, and items that are obtained from its own branch.

The next chapter will describe the details of how this is realized in the KRF type system, with the caveat that the presentation is a few years old and that there have been some marginal changes to it in order to bring the structure in line with the more recently developed ontology and taxonomy.

Chapter 4

The KRF Type System

In order to see how the KRF type system relates to the ontology, it is necessary to first understand the type system itself. Its purpose is to characterize the structure where *entities* have *attributes* and where attribute values can be arbitrary KR expressions, including sets, sequences, records and so forth. It shall specify what types of entities can have what attributes, and what is the permissible structure for an attribute value given the choice of attribute and the type of the carrier of the attribute. It shall also be able to express this for *all* entities that may occur in a software individual.

There is an unfortunate terminology conflict with respect to the term `entity` for the KRF ontology. From the point of view of the type system, entities are represented as symbols or composite-entity expressions, and they can have attributes. Numbers and strings are not entities, and sets, sequences are records are not entities although their elements can be entities. From the point of view of the ontology, on the other hand, it is natural to define `entity` as the overreaching concept at the top of the taxonomy, with `construct` as a relatively minor alternative to the side as shown above, but then one ends up with having the type of `action` as a subtype of `entity` although actions are represented as records and not as entities in the type-system sense.

The present chapter contains a description of the type system that was written in year 2008. A few details have changed since then, but the general structure is the same.

4.1 Signatures

An *entitystate* is a set of entities and their descriptions. A *signature* is an entitystate that is used to characterize the structure of other entitystates, called its *object entitystates*. The signature must always specify what types are used in the object entitystate and what attributes may be used by instances of a type. It may also provide other information, for example, what is the admitted structure for the values for a particular attribute.

A Very Simple Signature

Since signatures are entitystates, it makes sense to ask that a signature shall also have *its* signature. In order to avoid an infinite regress, it is desirable to have first of all a signature that can be used as a signature for itself. Consider first the following very simple example of a signature. We specify a number of entities and the attribute-value assignments for each of them.

```
Type             [type Supertype]
                 [has-attributes {}]
Supertype        [type Supertype]
                 [subsumed-by Type]
                 [has-attributes {subsumed-by has-attributes}]
Descriptortype   [type Supertype]
                 [subsumed-by Type]
                 [has-attributes {subsumed-by}]
Descriptor       [type Descriptortype]
Attribute        [type Descriptortype]
                 [subsumed-by Descriptor]
type             [type Attribute]
subsumed-by      [type Attribute]
has-attributes   [type Attribute]
```

This signature has the following property. For each entity e in the signature, one identifies the value t of its `type` attribute. Then, for every assignment to an attribute a of e, except when t = `type`, a shall be a member of the `has-attributes` attribute of t. Having this property is one of the requirements for being self-describing. As a matter of convention we omit `type` in the value for `has-attributes` since every entity must have a value for `type`.

The problem with this very simple signature is of course that it does not even begin to specify the permitted structures for attribute values. We shall add this soon below, but this extension requires introducing quite a number of auxiliary entities. This is because attribute values may be sets of entities, as we observe for the `has-attributes` attribute, and therefore we must introduce a way of characterizing such sets, and this again leads to a need for describing those set-expression characterizers. For this reason, we shall make another and much simpler extension before we turn to the attribute-value specifications.

Categories

In many applications there is a need to put a "flag" on some of the entities, for example for marking entities whose description needs further checking before it can be released. Rather than having to introduce one more attribute for each such flag, it is convenient to have a single attribute whose value is a set of applicable "flags". These flags are called *categories* in the KRF ontology. The following additional definitions are needed.

```
Category         [type Descriptortype]
                 [subsumed-by Descriptor]
                 [has-attributes {applicable-for}]
applicable-for   [type Attribute]
has-categories   [type Attribute]
```

The idea is that each category or "flag" shall be represented as an entity of type `Category` and that the `has-categories` attribute of an entity shall have as value the set of the categories that apply to that entity. Furthermore, each entity of type `Category` shall have an attribute called `applicable-for` whose value shall be a non-empty set of types for entities for which this category may apply.

Since `Category` has a `has-attributes` attribute, it becomes necessary to amend `Descriptortype`, `Descriptor`, and `Attribute` as follows.

```
Descriptortype  [type Supertype]
                [subsumed-by Type]
                [has-attributes {subsumed-by has-attributes}]
Descriptor      [type Descriptortype]
                [has-attributes {}]
Attribute       [type Descriptortype]
                [subsumed-by Descriptor]
                [has-attributes {}]
```

These amendments will anyway be needed in later steps, so they are not made only for serving the category concept.

In addition, if it should be desired to assign categories to `Descriptortype`, `Thingtype`, etc. then the description for `Supertype` must be amended as follows.

```
Supertype       [type Supertype]
                [subsumed-by Type]
                [has-attributes {subsumed-by has-categories
                      has-attributes}]
```

This addition is probably rarely useful in practice, but we include it here in order to provide additional work for the validation procedure to be defined below.

The Supersignature

We proceed now to a signature that also specifies attribute structure. This signature will be introduced in several steps. First, we repeat the same entity descriptions as above, including the amendments for categories, but introducing in addition an attribute for `Attribute` called `valuetype`. This attribute is going to be used for expressing the permitted structure for the value of the attribute in question.

```
Type            [type Supertype]
                [has-attributes {}]
Supertype       [type Supertype]
                [subsumed-by Type]
                [has-attributes {subsumed-by has-categories
                   has-attributes}]
Descriptortype  [type Supertype]
                [subsumed-by Type]
                [has-attributes {subsumed-by has-attributes}]
Descriptor      [type Descriptortype]
                [has-attributes {}]
Attribute       [type Descriptortype]
                [subsumed-by Descriptor]
                [has-attributes {valuetype}]
```

```
Category          [type Descriptortype]
                  [subsumed-by Descriptor]
                  [has-attributes {applicable-for}]
type              [type Attribute]
subsumed-by       [type Attribute]
has-categories    [type Attribute]
has-attributes    [type Attribute]
valuetype         [type Attribute]
applicable-for    [type Attribute]
```

Next, we extend the definitions of the entities of type `Attribute` so that they also have a value for `valuetype`. This value shall specify what is admissible structures for the value assigned to the attribute in question.

```
type              [type Attribute]
                  [valuetype Type]
subsumed-by       [type Attribute]
                  [valuetype Type]
has-categories    [type Attribute]
                  [valuetype (setof Category)]
has-attributes    [type Attribute]
                  [valuetype (setof Attribute)]
valuetype         [type Attribute]
                  [valuetype Type]
applicable-for    [type Attribute]
                  [valuetype (setof Type)]
```

The second requirement for self-description in a signature is as follows. Consider any assignment of a value v to the attribute a of an entity e in the signature, and let s be the value of the `valuetype` attribute of `a`. If v is an entity then `s` must either be equal to the `type` attribute of v, or it must be an element in the `subsumed-by` trail from the value of the `type` attribute of v. If v is a non-entity expression such as a set or a sequence, then it must conform to s according to the rules for every such kind of expression.

For example (writing henceforth $e.a$ for the value of the a attribute of e), above, `Attribute.subsumed-by` is `Descriptor` the type of which is `Descriptortype`; `subsumed-by.valuetype` is `Type`; this is accepted since `Type` is on the `subsumed-by` trail from `Descriptortype`. Also, `type.valuetype` is `Type` the type of which is `Supertype`; `valuetype.valuetype` is `Type`; this is also accepted for the similar reason.

Continuing with composite attribute values, `Descriptortype.has-attributes` is {`subsumed-by has-attributes`} and `has-attributes.valuetype` is `(setof Attribute)`. This is accepted according to the rule for the `setof` expression, since {`subsumed-by has-attributes`} is a set and each of its members has the type `Attribute`.

In order for the entire structure to be self-describing we need to introduce a type for operators such as `setof` (there will be more of them in the sequel) as well as a way for defining their structure. The following definitions do this.

```
Ecomposer         [type Descriptortype]
                  [subsumed-by Descriptor]
```

```
                [has-attributes {argtypes valtype}]
argtypes        [type Attribute]
                [valuetype Seq-type]
valtype         [type Attribute]
                [valuetype Type]
setof           [type Ecomposer]
                [argtypes (seq Type)]
                [valtype Set-type]
Set-type        [type Supertype]
                [subsumed-by Type]
                [has-attributes {}]
```

Much of this is self-explanatory. The value of the `argtypes` attribute shall be a sequence consisting of the types for the successive arguments of the composer in question, which is why we select its `valuetype` as `Seq-type`. The value of the `valtype` attribute shall be the type of expressions formed using the Ecomposer in question. (Notice the difference between `valuetype` and `valtype`).

This again requires us to introduce a definition of the operator `seqof` which is analogous to `setof`, as well as an operator `seq` of an arbitrary number of arguments, obtaining

```
seqof           [type Ecomposer]
                [argtypes (seq Type)]
                [valtype Seq-type]
seq             [type Ecomposer]
                [argtypes (seqof Type)]
                [valtype Seq-type]
Seq-type        [type Supertype]
                [subsumed-by Type]
                [has-attributes {}]
```

For example, `(seqof vehicle)` is the type for sequences of any number of elements where each element has the type `vehicle` or a type that is subsumed by `vehicle`. Similarly, `(seq person vehicle)` is the type for sequences of exactly two elements where the first element is a `person` and the second element is a `vehicle`. It is clear that the type for the argument-list for `seqof` is `(seq Type)`, and the type for the argument-list of `seq` is `(seqof Type)`.

With this, we obtain the following inferred descriptions for those composite entities occurring above:

```
(setof Category)    [type Set-type]
(setof Attribute)   [type Set-type]
(setof Type)        [type Set-type]
(seqof Type)        [type Seq-type]
(seq Type)          [type Seq-type]
```

This makes it possible to apply the self-description requirement even on those attribute values that are composite entities. For example, `applicable-for.valuetype` is `(setof Type)` whose type is `Set-type`, `valuetype.valuetype` is `Type`, and this is accepted since `Type` is on the `subsumed-by` trail from `Set-type`.

In order to verify the choice of `argtypes.valuetype`, consider for example the attribute assignment for `seq.argtypes` as `(seqof Type)`, the type of which is `Seq-type`. This agrees with `argtypes.valuetype` which is also `Seq-type` so the assignment is accepted.

Notice by the way the distinction that is made between *composite entities* such as `(setof Type)`, and LDX expressions that are not entities but e.g. sets or sequences, such as {`argtypes valtype`} or ⟨`Type`⟩. Entities whose type is `Ecomposer` are operators for forming composite entities, and composite entities have a type and other attributes just like atomic entities.

The actual supersignature that is used in Leonordo and CKL at present (April, 2008) contains the following additional entities in addition to those described above.

```
setofall          [type Ecomposer]
                  [argtypes (seq Type)]
                  [valtype Set-type]
setofsome         [type Ecomposer]
                  [argtypes (seq Type)]
                  [valtype Set-type]
join              [type Ecomposer]
                  [argtypes (seqof Type)]
                  [valtype Type]
```

Of these, `setofall` and `setofsome` are used like `setof` but provide additional information concerning whether the set in question shall be assumed to be the complete set or not; this has been described in the textbook. The `join` operation can be used in the `valuetype` attribute in order to form the union of several given types. It is not used in the supersignature itself, but its definition has been included there so that it is available for other signatures that are based on the supersignature.

Extensions to the Supersignature

The supersignature is self-describing in the sense that it is an adequate description of itself with respect to what attributes are used for which entity types, and what structure is admitted in attribute values. Signatures for applications may be thought of as a three-step structure consisting of the actual knowledgebase (K), the signature for the knowledgebase (S), and the signature for the signature which is the supersignature (SS). However, it is more fruitful to think of it as follows: S is a signature having the property that the union of S and SS is self-describing, and likewise K is a signature having the property that the union of K, S and SS is self-describing.

The advantage with this way of seeing things is that it is modularity-oriented: it makes it possible to build a library of signature modules which can be assembled according to need. Global library information specifies which modules depend on which other modules, and the union of a given set of modules is self-describing provided that for each module in the set, the set contains all the modules that the given module depends on (provided that the modules have been designed correctly and do not have any conflicts).

Self-description is important because it is a way of expressing type-checking. In the simple case of K, S and SS for a knowledgebase and its signature, the union of K, S and SS can only be self-describing if the contents of K

conform to the type information that is given in S and the signature S is organized according to the conventions that are represented in SS.

We proceed now to describing a few small signature modules that are often used.

The Scalar Signature

Most applications require the use of attribute values that are strings, numbers or other scalars. The signatures for such applications require the use of the appropriate scalar types, which are defined as follows.

```
Scalar-type   [type Descriptortype]
              [subsumed-by Descriptor]
              [has-attributes {}]
String        [type Descriptortype]
              [subsumed-by Scalar-type]
              [has-attributes {}]
Number        [type Descriptortype]
              [subsumed-by Scalar-type]
              [has-attributes {}]
Niltype       [type Descriptortype]
              [subsumed-by Scalar-type]
              [has-attributes {}]
Symbol        [type Descriptortype]
              [subsumed-by Descriptor]
              [has-attributes {}]
```

The absence of a value for an attribute is defined as equivalent to the attribute having the entity `nil` as value, and this entity is by definition the sole instance of the type `Niltype`.

The entity `Symbol` is included as a catch-all that can be used when an attribute has values that are implemented like entities but without any value for the `type` attribute.

The Signature for Records and Predicates

Besides sets and sequences, the LDX notation also allows the use of *records* which are written on the form e.g. `[R a b c]` consisting of a *record composer* `R` and an unspecified number of arguments. A record composer does not in itself specify the number and the types of the arguments, so when an attribute is going to have records as values then the `valuetype` of that attribute must specify any restrictions on the number or types of the arguments.

However, there is also the more specific case of *predicates* which are record composers that do specify the number and the type restrictions for their arguments. Records formed using predicates are called *literals* and are used to form logic formulas in LDX.

The following is the signature extension for records and predicates.

```
Record-type    [type Supertype]
               [subsumed-by Type]
               [has-attributes {}]
Rcomposer      [type Descriptortype]
               [subsumed-by Descriptor]
               [has-attributes {}]
record         [type Ecomposer]
               [argtypes (seq Rcomposer)]
               [valtype Record-type]
record-args    [type Ecomposer]
               [argtypes (seq Rcomposer (seqof Type))]
               [valtype Record-type]
Literal-type   [type Supertype]
               [subsumed-by Record-type]
               [has-attributes {}]
Predicate      [type Descriptortype]
               [subsumed-by Rcomposer]
               [has-attributes {argtypes}]
literal        [type Ecomposer]
               [argtypes (setof Predicate)]
               [valtype Literal-type]
```

For example, suppose `Email` is a record composer whose arguments are intended to be the `to`, `cc`, `Subject`, etc. fields in an electronic mail message. Suppose also that the value of the attribute `mail-exchange` is going to be a sequence of such records. The signature information for this is

```
mail-exchange [valuetype (seqof (record Email))]
```

If instead it is required that each record in that attribute shall only contain two arguments, one with the type `person` and the other with the type `String`, then the signature information should be

```
mail-exchange [valuetype
                 (seqof (record-args Email ⟨person String⟩))]
```

Finally, in a scenario where a university dean wishes to keep track of which of her faculty are consulting for which outside companies, the value of the `consulting` attribute for each department may be a set of literals of the form `[consults-for person company]`. This would be defined through

```
consults-for    [type Predicate]
                [argtypes (seq person company)]
consulting      [type Attribute]
                [valuetype (setof (literal {consults-for}))]
```

The argument for the operator `literal` is a set of predicates in order to allow for more than one predicate being used in a particular collection of literals.