KRF

Knowledge Representation Framework Project

Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Principles of Domain Modelling for Knowledge Representation

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-025, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/025/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2011/003/
Date of manuscript:	2011-01-07

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/

Foreword

The inspiration to write this lecture note came when the Reviews Editor of the *Artificial Intelligence Journal* asked me to write a review of the *Handbook of Knowledge Representation* which was then about to appear in print. Although I was duly impressed by the high quality and relevance of the Handbook's contents, I was also disappointed by the lack of common conceptual framework for the field as well as the duplication of materials and the failures to make use of fairly obvious cross-connections between the contents of different chapters. This criticism was also expressed in my review. However, since criticism without resulting action is merely annoying, I took this as challenge to show that it is in fact possible to organize the most important parts of Knowledge Representation, and therefore of Artificial Intelligence, in such a way that the different parts constitute a coherent whole.

The present manuscript represents the present state of that work. It is incomplete and the manuscript represents its first part, but I have decided to make it available anyway, partly in order to invite comments and feedback, and partly since this first part is used as background by another lecture note in the same series, namely the one called "Reasoning about Actions and Action Planning."

These lecture notes are part of a set of open-source lecture materials for a course in Artificial Intelligence ([1]) that use a common framework, notation and software. The notation that is used in the present lecture note is defined in logically preceding documents in that collection. It is also briefly described in the Appendix at the end of the present document.

[1] `http://www.ida.liu.se/ext/kraic/`

Chapter 1

A Generic Notation for Domain Modelling

Domain Modelling is the basic activity in Knowledge Representation. Virtually every application project requires a domain modelling activity where one decides how to represent the important information in that application. As a branch of the Knowledge Representation technology, Domain Modelling addresses systematic methods for such domain modelling activities.

Domain Modelling in Artificial Intelligence is mostly concerned with *qualitative models* consisting of relationships between entities of various kinds. In this respect it differs from modelling in e.g. control engineering where quantitative models are paramount.

Qualitative domain modelling is addressed not only in Artificial Intelligence but also in several other areas, such as the Database technology area within Computer Science, and in Computational Linguistics. Unfortunately there is no common basis for domain modelling in these disciplines; each one uses its own approaches. Making things worse, different subareas of Artificial Intelligence also do not use a common approach.

In spite of this diversity of formal or informal approaches, there are a number of important principles that recur in several of them. The present lecture note is an attempt to identify a generic common base within which major current approaches can be explained, and which can be used as a basis for continued lecture notes and continued work in several directions.

1.1 Building on Predicate Logic and Common Expression Language

Some approaches to domain modelling use first-order predicate logic as a basis, others introduce their own notation, and others again use more or less precisely defined graphical notations that may or may not be reexpressible as formulas. In the present treatise we shall use first-order predicate calculus (FOPC) as the formal basis, combined with the basic constructs of set theory such as sets and sequences that are specified by listing their elements. We

shall also use the typographical conventions of the KRF notation which have been introduced in preceding lecture notes in the present series.

Based on this essentially syntactic framework we define a generic base for domain modelling that combines the designs of conceptual graphs, the entity-relationship model, and those approaches to reasoning about actions and change that use the concepts of features and fluents. The resulting *generic domain modelling language* (GDML) is defined by introducing a number of predicates and functions in the framework of FOPC. In addition we shall use sets and sequences, as already mentioned, as well as *records* as an additional kind of construct besides sets and sequences.

The Generic Domain Modelling Language is a special case of the Common Expression Language (CEL) which uses the syntactic style of Knowledge Representation Expressions (KRE). The Appendix explains the relationships between the languages involved. Notice in particular that we shall stay entirely within first-order logic; we shall not use modal operators, nonstandard rules of inference, or any other similar variations of the basic formalism of logic.

1.2 Denotational versus Computational Evaluation

The lecture note 'List processing in the Knowledge Representation Framework' defined a number of predicates and functions that operate on sequences. These will be used in later parts of the present lecture note. There is however an important issue with respect to the *evaluation* of formulas, and this is an issue that is characteristic of Domain Modelling.

In order to understand this issue, please recall that both the semantics of logic and the semantics of programming languages define how to evaluate formulas in their respective formalisms. Consider now the following simple formula:

```
<mary peter rosanne>
```

From the point of view of logic it would be natural to decide that the value of this formula shall be a sequence of three elements each of which is a person, namely that person in the present context that has the name shown in the formula. There is therefore an abstract entity, namely, the sequence that has three elements, but each element in the sequence is a thing that exists physically, namely, the particular person of flesh and blood.

This consideration does not exclude other possibilities. In some contexts one might instead have decided that the person-names in question shall refer the respective person's passport, in a given pile of passports, or to the respective person's chair in a seating arrangement, or any other items according to the decision of the designer of the domain model. However, in many of these choices the value of a symbol such as `mary` in the example will be a physical thing.

From the point of view of programming languages, on the other hand, it is natural to restrict the perspective to software objects, that is, objects that can be created and manipulated by a program. In this case it will be natural to have a convention whereby symbols such as `mary` evaluate to

themselves but variables such as `.x` evaluate to a value that is specified in the evaluation environment. Therefore the expression shown above evaluates to itself, i.e. its value is the same as the expression itself, whereas the value of the expression

```
<mary .c rosanne>
```

in an environment where `.c` has the value `charles` will have the following value

```
<mary charles rosanne>
```

These are two different ways of evaluating formulas that are characteristic of logic and of programming languages, [1] respectively. Knowledge representation is characterized by using *both of these* modes of evaluation in an exchangeable fashion, and this is the basic concept for Domain Modelling. There is a need for the perspective of logic since domain modelling is about describing some aspect of the real world, but there is also a need for the perspective of programming languages since the chosen domain models shall eventually be processed by computers.

Different approaches to Knowledge Representation treat this duality in different ways. For example, the KIF representation language (Knowledge Interchange Format) is syntactically similar to the Lisp programming language, but it is defined with a logic-type semantics whereby KIF formulas can refer to the real world. Similarly, the Interagent Communication Language (ICL) is a language with a logic-type semantics that corresponds to the Logic Programming style of programming languages.

In the case of the Common Expression Language, the choice has been to use the same language for both purposes and to allow two different evaluation functions which we refer to as *denotational evaluation* in the spirit of logic and *computational evaluation* in the spirit of programming languages. The difference between them is that symbols (normally) evaluate to themselves in computational evaluation, and to a thing in the domain at hand in denotational evaluation.

1.3 Entities and Features

One of the first steps in the design of a domain model for an application is to identify particular things in the application with which one can associate various kinds of information. These things may be for example persons, houses, roads, plants or newspapers, to take some examples. For each such thing one needs to introduce an *entity* in the domain model. The entity may be a single symbol or a composite expression.

We shall use the term 'entity' for the symbol or expression that represents a particular thing in the application. The thing that the entity refers to will be called its *designation* according to the domain model. Composite entities using entity-formers are defined in the syntax for Knowledge Representation Expressions.

[1] At least for A.I. style, interpretive programming languages. For other languages it remains that the value of an expression must be a software object, but the value of the expression is not necessarily equal to the expression itself.

Entities may need to be introduced during the operation of a computational system. Consider a domain model for a kitchen where there are a number of knives and forks in a drawer. One may like to define an action for "obtain one fork from the drawer" which results in the creation of an entity for the fork obtained, but it is not desirable to have separate identifiers for all the forks in the drawer already at the start of the computational session, and besides when defining this action it will not be possible to decide which of the fork entities should be obtained. Therefore the action of witdrawing a fork must be defined so that it creates an additional entity.

Predicate logic makes it possible to introduce domain-specific predicates and functions for expressing the information about entities in the domain model. For example, in an application where one has introduced entities designating a number of houses, with a separate entity for each of them, one might also introduce a function `color-of` and use it so that if `house-4` is the entity designating one particular house, then the value of the term `(color-of house-4)` will be the color of that house. Notice then how this expression is evaluated denotationally and computationally. Denotationally, the value of the symbol `house-4` will be the particular, physical house, and the function `color-of` is a mapping from actual houses to actual colors. It is assumed then that colors are abstract objects that can occur as arguments and values of functions.

The definitions for computational evaluation, on the other hand, are most likely set up so that the symbol `house-4` evaluates to itself, and the function `color-of` is a mapping from symbols representing houses to symbols representing colors, for example the symbol `white.` The definition of the function `color-of` will then consist of looking up the color of the house in question in the computational system's database or knowledgebase.

The *correctness* of a domain model will be defined in due time, but it can be observed already now that it must be defined in such a way that in a correct domain model the results of the two kinds of evaluation are consistent with each other.

Expressing domain models directly in terms of domain-model-specific functions and predicates is often quite inconvenient, and there are a number of constructs that can be defined within the framework of first-order logic and that facilitate the job of designing qualitative domain models. We shall focus on two kinds of constructs, namely *features* and *relationships.*

A feature is an abstract entity that may be assigned a value using a separate predicate, often called the *Holds* predicate. We shall write this predicate as `H` if it has a time argument and as `Hc` if it does not. For example, rather than writing

```
[= (color-of house-4) white]
```

one may write

```
[Hc (color: house-4) white]
```

where `(color: house-4)` is a feature and `color:` is therefore a feature-valued function. The predicate `Hc` takes a feature as its first argument and specifies that the value of that feature is what is given as the second argument. One advantage of this arrangement is that it makes it possible to express statements like "Mary knows what is the color of `house-4.`" In

this case it is the feature `(color: house-4)` that is the object of Mary's knowing.

The use of features is particularly convenient for dealing with information that changes over time. In this case one uses the predicate `H` that has three arguments. The first argument is a timepoint and the other two arguments are like for `Hc`. For example, to say that the color of the house `house-4` is white on January 1, 2012, one might write

```
[H tp.2012-01-01 (color: house-4) white]
```

The following notation is used for timepoints in the exammples in the present text, although it is not suggested for general use: `tp.2012-01-01` designates the date in question viewed as a single timepoint, so that January 2, 2012 is the next timepoint. By comparison, `ti.2012-01-01` designates the same date but viewed as a time interval consisting of a sequence of timepoints.

If one were to express the change of color over time using the function `color-of` then one would need to provide it with an additional argument for expressing the timepoint, and the same would be true for every other function where the value depends on time. By using features and the `H` predicate one can concentrate the identification of time to one particular predicate.

The predicate `Hc` is intended for use in cases where the applicable timepoint is defined by the context. For example, the specification of the preconditions for an action usually apply to the timepoint where the action starts, so the preconditions for actions using a particular verb can often be expressed using an expression that uses the `Hc` predicate and that refers to the various components of the relationship expression for the action. However, the use of this predicate involves some additional considerations, so it will not be used any further in this chapter; we shall return to it in Chapter 2.

1.4 Relationship Expressions

The other important construct is the *relationship.* Relationships may be used for a variety of purposes, but one major usage is for counterparts of simple full sentences in natural language. Consider for example the sentence

```
John will travel by bus from Stockholm to Uppsala
```

This phrase may be rendered as the following relationship

```
[travel :by John :using bus :from Stockholm :to Uppsala]
```

Notice therefore that a relationship is a *composite expression* in our terminology; it is an expression that consists of a number of parts, just like a set or a sequence is an arrangement of a number of parts. This expression that we call a relationship *does not in itself make any statement,* it is just a description of a phenomenon that may or may not actually exist. In order to *state* that John will travel by bus from Stockholm to Uppsala tomorrow, one should write

```
[W ti.2012-01-01 [travel :by John :using bus
                        :from Stockholm :to Uppsala ]]
```

provided that the trip is made on January 1, 2012. The predicate `W` may be read as "within" and specifies that there is an action of the kind described by the second argument that occurs within the time interval given as the first argument.

The use of tags in the notation for relationships is not logically necessary, but quite convenient. It is true that one could write the parameters as arguments for example

```
[travel John bus Stockholm Uppsala]
```

where each argument position is dedicated to a particular purpose. The use of tags offers the following advantages:

- The mnemonic advantage that the tag reminds the reader of the purpose of each of the arguments
- If some of the parameters are unknown then the formulation without tags needs to introduce existential quantifiers, for example as shown below
- If tags are selected so that they have a semantic content, for example tags for the *actor* and the *instrument* of an action, then it may be possible to write axioms that apply to entire groups of verbs and that allow inference from specific occurrences of an action.

The following example illustrates the second point:

```
[exist .c [travel John bus .c Uppsala]]
```

compared with the expression using tags where parameters may be omitted if the value is unknown:

```
[travel :by John :using bus :to Uppsala]
```

One use of relationships is for representing *actions* which are usually counterparts of simple natural-language sentences involving a verb and its dependent phrases, like in the example above. However, there are also many other uses of relationship expressions. For example, a date in the Gregorian calendar might be written as

```
[Gdate :year 2011 :month 01 :day 05]
```

Other kinds of descriptions may be expressed in similar ways, for example geographical locations expressed by latitude and longitude in degrees, minutes and seconds, or street addresses, or even personal names (allowing for regional differences for how to form person's names).

1.5 Component Models and Process Models

Qualitative models for domains involving change and the passing of time are of two major types: *component models* and *process models.* In component models the domain is characterized as a collection of entities, a set of features for each entity, the values of the features at each point in time, and constraints that specify the interdependencies between these features. For example, a simple component model for a room in a house may contain entities for a particular lamp and a particular electric switch; one feature can

represent whether the lamp is on or off, and another feature can represent whether the switch is in the on or off position. There can be a constraint that specifies whether the lamp is on or off as a function of the position of the switch.

In process models, on the other hand, one uses entities, features, and constraints in the same way as in component models, but in addition the domain model allows the use of *processes.* Each process is active during a period of time and involves some of the features; it influences the values of some of them, usually depending on the values of some other features at the same time.

Processes can be described on two levels of detail. The *operational level* specifies constraints between the features of the process at each timepoint in the duration of the process; the *effect level* specifies or restricts the values of the features at the end of the process' duration as a function of their values when the process started. The use of the effect level is quite important since it makes it possible to predict the effect of a sequence of actions (= processes) without having to consider the details of each of them, and therefore it is the basis of action planning. However, the operational level is also important, in particular in robotic systems where it is the basis for the design of the controllers that are needed for controlling the processes.

Process domain models are used in several branches of Knowledge Representation, in particular for action planning, in cognitive robotics, and for reasoning about mechanical devices. The word 'process' is used by the latter area whereas in temporal reasoning and reasoning about actions one usually prefers the word 'action' or 'event'. None of these terms is ideal. The problem with the word 'event' is that it is also extensively used for a momentary change of value of one feature. The word 'process' is fine for mechanical systems but seems awkward if one applies it to things like John's trip to Uppsala. The word 'action' is fine for things that are performed by persons or animals, but it is instead awkward for processes that arise by causation, for example a snowstorm, a car accident that is caused by that snowstorm together with the negligence of the driver, or the resulting traffic jam.

For the present purpose we will use both the words 'process' and 'action' and define them so that a process is an instance of an action. Thus the expression

```
[travel :by John :using bus :to Uppsala]
```

represents *one action;* each time that John makes such a trip is *one action instance,* and a process is the same thing as an action instance. Actions that do not have an animate agent, such as

```
[volcano-eruption :at mount.etna]
```

will be referred to as "actions by nature." Notice the use of the tag `at` rather than `by` here, since the tag `by` is reserved for indicating the agent.

1.6 Preconditions and Effects of Actions

Both component models and process models use *constraints* for characterizing the dependencies between the values of different features; process models also use them for characterizing the dependencies between the occurrence

of processes and the values and change of values of features, and even for the causation from one process to another.

In order to write these constraints one needs a predicate that specifies that a particular action instance, or process, takes place during a particular period of time. The `W` predicate that was used for an example above is not the simplest possible one. The primitive predicate is `D` that is used as

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

for stating that a process that is an instance of the action `.a` begins at time `.s` and ends at time `.t` The action `.a` is a relationship expression formed in the way that has been described above. If this statement holds then the proposition `[W .i .a]` holds for every interval of time that contains or is equal to `[ivl .s .t]` i.e. the closed interval of time between `.s` and `.t.`

There exist minor variations to this notation, for example using a predicate similar to `H` but having two arguments, namely a timepoint and a truth-valued feature, for expressing that the feature has the value `true` at the timepoint in question. Some authors do not use the predicate `D` and consider each verb as a separate predicate with timepoints or an interval among the arguments.

In order to draw effect-level conclusions about the occurrence of a process one needs *action effect axioms* which are typically formed as an implication with a literal for the predicate `D` among the antecedents, as in the following example.

```
(imp [D .s .t [paint :obj .h :as .c]]
     [H .t (color: .h) .c] )
```

This axiom says that if the object `.h` is painted with the color `.c` during a time interval from `.s` to `.t` then the color of that object at time `.t` is `.c.`

However, action effect axioms must be complemented with *precondition axioms* that specify the conditions that must be satisfied for the action to be feasible at all. As an example, suppose we wish to specify just one condition for the `paint` action, namely, that the person that does the painting "possesses" paint with the color in question at the starting time of the action. We may now adjust the notation so that it is the paint, rather than the color of the paint that is the parameter of the verb `paint,` and we need to use the function `color-of` for the bucket of paint. The revised effect axiom is

```
(imp [D .s .t [paint :obj .h :with .p]]
     [H .t (color: .h) (color-of .p)] )
```

Preconditions are specified using the predicate `P` that takes a timepoint and an action as argument, and that states that it is possible to start execution of the action at the given time. The letter `P` may be read as "precondition" or as "possible." The simple precondition for the `paint` action can be written as follows:

```
(imp [H .t (possesses .a) .p]
     [P .t [paint :by .a :with .p]] )
```

This is a very simplified example, of course, and for less simplification one would need several additional preconditions, beginning with one stating that `.p` is in fact a bucket of paint.

Continuing the same example, if one actually wishes to use the color rather than the paint as the parameter of the painting action, still using the tag `with` then one would have to write the precondition expression as

```
(imp [exist .p (and [H .t (possesses .a) .p]
                    [= (color-of .p) .c] )]
     [P .t [paint :by .a :as .c]] )
```

1.7 Ground Models and Entity Descriptions

Although in principle it would be possible to express everything using the predicates `H` and `D,` in practice it would be quite inconvenient to do so. There are some basic kinds of information that are better expressed using additional predicates. In particular there is a need for a predicate `hastype` which may be used like in

```
[hastype house-4 house]
```

in order to specify what is the type of the entity `house-4,` or

```
[hastype bucket-4 bucket-of-paint]
```

with obvious meaning. The type is in turn an entity that also has a type, and so forth; in this generic representation we assume that every entity has a unique type.

This representation is only adequate if we can assume that the type of something does not change over time and that it is not the subject of knowledge or belief. This means that the type concept shall only be used for basic distinctions, like distinguishing between persons, buildings, fruits, and so forth. It shall not be used for more detailed distinctions, like "house where someone is living" versus "deserted house," since this may change over time and may be subject to incorrect belief.

Information such as this restricted type information will be called *static information,* and a rule of thumb for the design of domain models is that for static information it is appropriate to introduce domain-specific predicates, similar to `hastype,` but for other information it is preferable to use features together with `H` or other, similar predicates.

The decision that a particular kind of information is static is often due to a simplification that is made by the designer of the domain model. For almost every kind of fact that one can think of, it is possible to imagine situations where this fact changes over time, or that there are situations where someone has a mistaken belief about that fact. However, one important aspect of practical domain modelling is that one has to restrict the generality of the representation to what is needed in practice.

It is common that a domain model contains a considerable number of ground literals that are formed using the entities in the model, that is, statements that are formed using a static predicate, or the negation of one, together with explicitly given entities as arguments. The set of these literals will be called the *ground part* of the domain model or more simply as the *ground model* for the application at hand. (Ground formulas are those that do not contain any variable).

The use of entity descriptions may be seen as a practical way of storing and managing ground models. (Entity descriptions and their aggregation into entityfiles are described in the lecture notes "KRF Overview" and "Managing Information Aggregates in the Knowledge Representation Framework.") In simple cases, for a given binary static predicate `R` and a given entity `e` it may be convenient to represent the set of all literals of the form `[R e ci]` for different entities `ci` as a set {`c1 c2 ... ck`} that is assigned as the value of the attribute `R` for the entity `e`. More complex structures for the attribute value may be needed in particular situations, for example for predicates of more than two arguments. The ground model will then be represented by the collection of all the entity descriptions for the entities in the domain model.

1.8 Reification of Relationship Expressions

A relationship is merely an expression: it is an aggregation of a number of components, but it does not have any inherent significance besides this. Every domain model will use its own set of operators for forming these expressions (such as `travel` and `Gdate` in the examples above) and its own set of conventions for the order of arguments and the use of tags in them. The *meanings* of these operators and tags must be defined by *domain axioms* that specify what conclusions can be drawn from given statements.

Since predicate logic is the overall framework, those given statements must be expressed as propositions in the sense of logic, with ground literals as the important special case. Relationship expressions are terms, and not literals; verbs and other operators that form relationship expressions are not predicates. Therefore it is not possible to draw any conclusions from relationship expressions in themselves, but only from literals that are formed using predicates and where relationships occur as arguments.

This is quite sufficient in many cases, but there are also situations where one wishes to *reify* a relationship, that is, to introduce an entity that *designates* the phenomenon that the relationship *describes.* In the example relationship expression

```
[travel :by John :using bus :from Stockholm :to Uppsala]
```

one can imagine situations where it is desired to specify additional information about this particular trip, for example, what were its consequences, how was it witnessed and reported, and so forth. It would not be appropriate to express that information using additional literals where the relationship expression occurs as one of the arguments since that would be ambigous: John may have made several bus trips from Stockholm to Uppsala.

The solution is instead to let the computational system introduce an additional entity with a previously unused name, for example `travel-142` for the trip in question, and to use that entity for those additional statements. It will then be up to the system to assure that different trips are given different names, and that (to the largest extent possible) the same name is used every time one specific trip is being considered. Such an entity is called a *reification* of the given expression.

There must of course also be a proposition that specifies the relation between the reification and the relationship being reified. This may be done using a

single predicate `designates,` for example

```
[designates travel-142 [travel :by John :using bus
                            :from Stockholm :to Uppsala ]]
```

However another possibility is to convert each tag in the relationship to a binary predicate, and to write the defining information for the reification as follows

```
[hasverb travel-142 travel]
[by travel-142 john]
[using travel-142 bus]
[from travel-142 Stockholm]
[to travel-142 Uppsala]
```

The binary predicates that are introduced in this way are static predicates since they do not admit any change over time and there is no reason for them to do so. If one does not wish to introduce many separate predicates of this kind then one alternative is to introduce a single, ternary predicate for this purpose, so that the translation becomes

```
[hasverb travel-142 travel]
[param travel-142 by john]
[param travel-142 using bus]
[param travel-142 from Stockholm]
[param travel-142 to Uppsala]
```

The representation using several different, binary predicates is appropriate if one is using a computational framework that precisely supports binary predicates, such as description logic. The representation using a single, ternary predicate is instead appropriate if one's software does not have any particular preference for binary relations, and if one is able to state properties that apply for the `param` predicate in general or for specific choices of its second argument.

A crucial question concerning such reified relationships is whether they shall be considered to be unique or not. One possible view is that the reification of an action (actions being a kind of relationship) is an action instance, i.e. a process, which means that one and the same relationship may have several reifications. [2] Another possible view is that the reification is unique and that it stands in a one-to-one relationship with the relationship expression. We leave this question open for now.

1.9 Featurization

Early in this chapter we described the introduction of features as a way of creating a representation where one can characterize changes over time and belief about specific facts. (This is a special case of belief, and more general cases of belief require another kind of representation). Given what has been said about static information, relationships, and the reification of relationships, it is natural to ask what are the possibilities if one wishes to have entities that offer the same possibilities as features but which apply to static literals and to relationships.

[2] There is an issue whether it is appropriate to call them reifications in this case, but we leave that aside.

Notice first of all, then, that if an example like "the color of `house-4` is white" is considered as static information then it may be expressed using a binary predicate `has-color` as

```
[has-color house-4 white]
```

The shift of representation from there to the use of a feature, in

```
[H tp.2012-01-01 (color: house-4) white]
```

can be considered as setting a pattern for how all binary static predicates can be *featurized.*

Suppose for example that we are working with a domain model where the relationship between a person and his or her father is represented statically, so that there is a binary predicate `has-father` whereby one can write

```
[has-father rosanne stephen]
```

and then occasionally there is a reason to make statements about someone's belief about who is the father of Rosanne, including the possibility of incorrect beliefs. This can be done by introducing a feature-valued function that corresponds to the predicate `has-father` – say `father:` – and constructing the feature `(father: rosanne)`. If a person called Charles believes that a person called Bill is Rosanne's father then this belief may be represented as the following relationship

```
[believes :by charles :re (father: rosanne) :val bill]
```

This relationship can of course be used in the ways described above, such as stating it as the argument of the `D` predicate, or reifying it.

In these examples we have seen the need for introducing a feature-valued function that corresponds to a particular monary function or binary relation for static information. In a practical system one will probably prefer to use a systematic naming convention for going between one and the other.

The featurization method is also applicable to the components of reified relationships. Consider the example above where the relationship

```
[travel :by John :using bus :from Stockholm :to Uppsala]
```

was reified as the entity `travel-142`. If Rosanne knows about this trip but believes that John took the train instead of the bus then this could be expressed as follows using the approach shown above:

```
[believes :by rosanne :re (using: travel-142) :val train]
```

where again of course this relationship should be used as the argument of a `Holds` -type predicate in order to be asserted.

1.10 Representing Part-Whole Relationships

The relation that holds between a physical object and each one of its parts is important in very many domain models. In principle it is very easy to represent it: we simply need a predicate `is-part-of` between the part and the whole. It is natural to consider this as a transitive relation since it is not always possible to specify an "immediate part of" relation in terms of layers of decomposition.

It is also natural to use a predicate `is-separate` that holds between two objects if neither of them is a part of the other and they do not have any common parts. The negation of this predicate may be used for describing connectors in the sense of parts that are partly inserted into two or more other, separate parts.

These predicates are easy to define and to use if they represent static information. However, there is obviously a problem when an application allows for objects to lose some of their parts, or to acquire additional parts that are added to them. In fact there are two problems, the first one being to answer the question at what point an object ceases to be "the same" object if one part after the other is replaced in it. Some answer must be given to this question if the representation described here is used, since we have been assuming that each object is represented by a particular entity, so if an object is considered to have become another object by the replacement of parts then another entity must be introduced in the representation.

One natural solution to this problem is to identify some indivisible part of the object as the one providing the identity, so that other parts may be replaced while retaining the same entity as name for the object. Another approach may be to consider *every* replacement of a part as the destruction of the old object and the creation of a new one. This requires however that there are ways of keeping track of the relations between the successive "objects" that result from this convention.

The second question occurs once it has been determined that there is an addition, removal or replacement operation on some part of a given object and it has been decided to still consider it as the same object. How shall one then represent the change in the `is-part-of` relation? It is natural to look for featurization for an answer, that is, to rewrite ground literals such as

```
[is-part-of muffler-127 car-19]
```

as

```
[H .tp (is-part-of: muffler-127) car-19]
```

whereby the information concerning what object `muffler-127` is part of is made subject to change over time and to knowledge and belief.

This representation has some computational consequences. In the original representation using a binary predicate `is-part-of` it is natural to implement this predicate using two-way links between parts and wholes, so that it is easy to obtain the set of parts of a given whole object. In the featurized representation, one may have to only store the link from the part to its immediate superpart (that is, the smallest part that it is stated to be a part of) using propositions like the one just shown, and to reconstruct the current set of parts for a given whole each time this information is requested. However, the question of efficient implementation in terms of datastructures for the kind of information discussed here is a nontrivial topic in itself; we shall disregard it at this point and focus only on the modelling issues.

Part-whole relationships may seem to be a simple kind of information but this impression is deceptive and there are in fact many possible complications. The topic of qualitative modelling of mechanical systems has to address this problem to its full extent.

1.11 Representing the Creation and the Destruction of Objects

Returning to the entity `house-4` in previous examples, consider the action of destroying that house so that it does not exist any longer. If a domain model shall allow the representation of such events then it must make it possible to specify a *time of destruction* for each entity. Similarly, since there may be actions for building a house, or other actions that result in the creation of a new object, there must be a possibility of specifying the *time of creation* of an entity.

We shall use the binary predicates `created-at` and `destroyed-at` for this purpose. They are of course static predicates, but in some applications there may be a need for featurizing them in order to represent alternative beliefs about the time of creation or destruction.

Several points of view are possible with respect to the times of creation and destruction of reified relationships, and in particular for reified actions. One possible view is that the starting-time and the ending-time of the action are the times of creation and destruction of the reification.

1.12 Overview of Related Approaches

Qualitative Domain Modelling is traditionally addressed in several branches of Artificial Intelligence as well as in neighboring disciplines. The following is a brief overview of contexts where it appears and is addressed.

1.12.1 Conceptual Graphs

Conceptual graphs (CG) were introduced by J. Sowa [see his article in the *Handbook of Knowledge Representation*] as a way of representing the contents of sentences in natural language. Sowa distinguishes between Core CG, Extended CG and Research CG. Extended CG is a typed superset of the core, and Research CG is a family of extensions that are being studied.

Core conceptual graphs are presented as graphs rather than as formulas but in essence they are similar to relationship expressions for simple natural-language sentences as described above. One difference is that each parameter in the expression may contain both a type entity and an entity representing an instance of that type. The former is obligatory whereas the latter is optional. With an ad-hoc adaptation of the notation that was used above one could then have written the travel example as follows:

```
[travel :by person/John :using bus
          :from city/Stockholm :to city/Uppsala]
```

Sowa describes how a conceptual graph such as this one may be translated to first-order logic; the translation is similar to the reification operation described above and may have the following result, using our notation:

```
[hasverb travel-142 travel]
[by travel-142 john]
[hastype john person]
```

```
[exist .b (and [using travel-142 .b]
               [hastype .b bus] )]
[from travel-142 Stockholm]
[hastype Stockholm city]
[to travel-142 Uppsala]
[hastype Uppsala city]
```

The parameter for the 'instrument' only contains the type and not the instance and therefore the translation introduces an existentially quantified expression for it. It would also be possible to introduce an additional entity, for example `bus-19`. Conversely it is also possible to use an existentially quantified variable instead of the new entity `travel-142`, as follows:

```
[exist .t (and
    [hasverb .t travel]
    [by .t john]
    [hastype john person]
    [exist .b (and [using .t .b]
                   [hastype .b bus] )]
    [from .t Stockholm]
    [hastype Stockholm city]
    [to .t Uppsala]
    [hastype Uppsala city] )]
```

This is adequate as a translation of the relationship expression (assuming a sufficiently loose interpretation of the 'existence' of the entity denoted as `.t`) but of course it loses the possibility of referring to a reification for the relationship.

There are many other aspects to the theory of conceptual graphs, so the present can only serve to give a general idea of how they are related to the generic modelling concepts.

1.12.2 The Entity-Relationship Modelling Technique

The use of *entity-relationship diagrams* is a widely used technique in software engineering and database system design. This technique is usually referred to as the "entity-relationship model" but we would like to avoid that use of the word "model" and prefer to call it a technique.

Entity-relationship diagrams are intended to be used in an early stage of information system design, namely, for requirements analysis where they are used for describing information needs and the inherent structure of the information that is eventually going to be stored in, for example, a database. In a later stage of the design process, these diagrams are converted to other forms that are customarily referred to as the "logical design" and the "physical design." However these need not concern us here, and we shall restrict the attention to the entity-relationship diagrams as such.

The basic design for entity-relationship diagrams addresses static structures and does not mention the issue of change of state in the model being designed. This viewpoint is natural if one considers that the current contents of a database shall always be a model of the current state of the application, and if there is no need for representing the past history of that application. Under these assumptions, change of state in the application is implemented

as a corresponding change of state in the database, and the database contents do not explicitly represent earlier states or processes of the application in a systematic way.

Three notions are of basic importance for the entity-relationship diagram technique, namely entities, relationships, and attributes. Entities and relationships are used in essentially the same way as in the Generic Domain Modelling Language, with minor differences. One difference is that a strict distinction is made between entities and entity-types, whereas in the GDML and the CEL the type of an entity is again an entity, recursively.

Attributes in entity-relationship diagrams are used for assigning values to entities and to relationships. They correspond to the static binary predicates of the GDML, except that attribute values are required to be scalars, such as numbers or strings. A static relation between two entities must be represented using a relationship rather than as an attribute assignment.

In entity-relationship diagrams it is possible to assign attributes to relationships; this is analogous to introducing additional static literals with the reification of a relationship as one of the arguments.

However, entity-relationship diagrams do not depict entities and relationships per se, but sets of entities and sets of relationships. This is due to their use in the requirements phase for the design of large systems.

Consider again the travel example from above. In an entity-relationship diagram it would rather be written as follows

```
[travel :by person :using bus :from city :to city]
```

so that the diagram is only a schema that admits several instantiations using different instances of the types `person, bus,` and `city.` The original, graphical representation of the diagram [3] uses nodes for the main symbols and arrows for the relationships between the nodes. However, these arrows are decorated with additional information so that they can indicate *participation constraints.* For example, the following relationship [4]

```
[has-father :by rosanne :val stephen]
```

will be an instance of the following entity-relationship diagram

```
[has-father :by person :val person]
```

and the graphical representation for this diagram allows one to express that each instance of the type `person` must be related to *exactly one* instance of this diagram in its `by` component.

This is not the place for developing one more notation for entity-relationship diagrams, but just to make the idea with participation constraints concrete we show an example of how one might write the reified version of this entity-relationship diagram without and with participation constraints. Let `father-schema` be the name of the diagram for `has-father` relationships, and write it in reified form as follows.

[3] There are many textual notations for entity-relationship diagrams but this is not one of them. We have taken the liberty of rephrasing the diagram in a way that is consistent with the Generic Domain Modelling Language.

[4] Recall that attributes must have scalar values in entity-relationship diagrams, so the relation between a child and its father must be represented using a relationship expression.

```
[verb father-schema has-father]
[alpar father-schema by person]
[alpar father-schema val person]
```

The predicate `alpar` will then stand for "allow parameter" and specifies that instances of `father-schema` may have parameters for the tags `by` and `val`. In order to add the participation constraint one may modify the formulation as follows:

```
[verb father-schema has-father]
[alpar-x1 father-schema by person]
[alpar father-schema val person]
```

The predicate `alpar-x1` implies the predicate `alpar` for the same arguments, but in addition it specifies that each instance of the type given as the third argument must participate in *exactly one* instance of the entity-relationship diagram given as the first argument.

The meaning of `alpar-x1` in terms of `alpar` can then be partly characterized using the following axiom:

```
(imp [alpar-x1 .s .tag .p]
     [all .e (imp [hastype .e .p]
                  [exist .r (and [is-instance .r .s]
                                 [param .s .tag .e] )])])
```

1.12.3 Reasoning about Actions and Change

The area of reasoning about actions and change studies the logic for characterizing actions and processes. The use of features (called fluents by some authors) and the `H` and `D` predicates for characterizing the preconditions and the effects of processes are basic to the area. Continued and more complex questions for this research include the treatment of concurrent actions, chains of cause and effect, continuous change, actions with alternative outcomes that may be associated with probabilities, and others more.

There are two major approaches to reasoning about actions and change, namely representations with explicit time, and the use of the situation calculus.

Explicit-Time Logic

The representation that was used above is based on the notion of using the current time as an explicit argument for predicates that characterize actions and change. Usually it is assumed that the time axis consists of the natural numbers from 0 and up, or that it is isomorphic to this domain of numbers, but the formalism as such also allows the use of continous time or a forward-branching time domain.

The use of this representation for actions and change for the purposes of Knowledge Representation was first proposed by Y. Shoham in his 1986 article at the ECAI conference [reference] It has also been adopted by other researchers (more or less independently of Shoham's proposal), in particular by R. Kowalski, M. Shanahan et al. for the so-called *Revised Event Calculus,* by E. Sandewall for the *logic of Features and Fluents,* and by P. Doherty for

Time and Action Logic. The difference between these approaches lies mostly in the choice of nonmonotonic inference method for the logic in question. Shanahan uses an inference scheme that is based on Logic Programming, Doherty uses the PMON inference method, and Sandewall's contribution was to define an underlying semantics and to analyze a number of such inference methods and to identify the range of applicability of each of them.

The work in the Sandewall-Doherty tradition and in the Kowalski-Shanahan tradition are often described separately from each other, but in fact their semantics is the same and the differences are only in the choice of nonmonotonic inference methods and in the introduction of extensions in various directions. It is therefore natural to treat them together. This requires a common and neutral name, and we propose to use the term *Explicit-Time Logic for Actions and Change,* or just *Explicit-Time Logic* for this purpose.

The Situation Calculus

The Situation Calculus was originally proposed by J. McCarthy around 1960, although the presently used variant of it is due to R. Reiter and H. Levesque in their work starting around 1985 *[Check exact time and reference for this.]* It was therefore introduced prior to Explicit-Time Logic, but in retrospect it is more natural to consider Situation Calculus as a more specialized technique.

The basic idea in Situation Calculus is to use a forward-branching time axis where each "timepoint" has one successor for each of the actions that can be performed starting in that timepoint. These generalized timepoints are called *situations,* starting with an initial situation that is customarily written `S0`. If `.s` is a situation and `.a` is an action then `(succ .s .a)` is a successor of `.s`. Situations are used as the first argument of the `H` predicate, and the `D` predicate is not used.

There are two reasons why this should be considered as a specialized technique compared to Explicit-Time Logic. First, it is more restricted from the point of view of expressivity since it is not suitable for characterizing concurrent processes and it even has difficulties with nondeterministic actions. Also, Situation Calculus can be embedded in Explicit-Time Logic merely by introducing the successor function `succ` and the following axiom that characterizes it completely:

```
[all .s [all .a [D .s (succ .s .a) .a]]]
```

Every effect law that has been expressed using the `D` predicate and in the ways shown above can be used in a Situation-Calculus context as well using this axiom. Consider for example the following simple effect law for the verb `paint`

```
(imp (and [D .s .t [paint :by .a :obj .h :with .p]]
          [H .s (possesses .a) .p]
          [= (color-of .p) .c] )
     [H .t (color: .h) .c] )
```

It is easily seen that if `s4` is a situation and the following propositions are known

```
[H s4 (possesses john) paint-bucket-12]
```

```
[= (color-of paint-bucket-12) white]
```

then it follows

```
[H (succ s4 [paint :by john :obj house4 :with paint-bucket-12])
   (color: house4)
   white ]
```

The most attractive property of the Situation Calculus is that it provides a simple and powerful method for regressive planning, that is, for a planning process that starts with the desired goal and that builds a plan "backwards" towards the current state of the world.

1.12.4 Modelling of Physical and Technical Systems

The modelling, representation and reasoning about physical and technical systems has its origin in the modelling of physical devices, for example electronic circuits, or mechanical devices such as mechanical clocks. This area has developed strong methods that turn out to also be applicable for many purposes outside their original intended range.

Three major approaches are used for domain modelling in this area: component models, process models, and field models. The first two of these have already been described. Field models have a different character; we cite from the article by K. Forbus in the *Handbook of Knowledge Representation:*

Both component and process ontologies are forms of what are called lumped parameter models. Many important phenomena, however, such as weather patterns and phase portraits, are spatially distributed, and cannot be understood without reasoning about that spatial structure. Field ontologies represent that structure by dividing space into regions where some parameter of interest takes on qualitatively equivalent values. This space can be physical space, e.g., for reasoning about heat transfer or meteorology, or phase space, e.g., for reasoning about dynamics, or configuration space, e.g., for reasoning about mechanical systems.

Field models fall outside the scope of the present lecture note.

1.12.5 Other Important Related Approaches

Other related approaches include representations that are used in *Computational Linguistics* and those that are used in the research for the *Semantic Web.* It is intended to add material about these here, as well as additional text about the modelling of physical and technical systems.

Appendix: Details of the Notation

The primary purpose of the present lecture note is for use in a university course on Artificial Intelligence, and a second purpose is to demonstrate that it is possible to present several central A.I. techniques in a much more unified way than what is usually done. These purposes have required the introduction of certain notation, but not to the point where every detail of the syntax has to be specified.

An additional purpose is that later lecture notes in our course materials shall be able to use not only the concepts, but also the notation that is introduced here. This purpose requires more precision. First of all, it is necessary to distinguish between those predicates and functions that are intended to be included in the basic notation, and on the other hand those predicates and functions that have merely been used for some of the examples in the earlier chapters. Furthermore, there should at least be a beginning towards specifying axioms and/or an underlying semantics in order to clarify the meaning of the various parts of the basic notation. The present Appendix is dedicated to this purpose.

Language Levels

The notation described here is called the *Generic Domain Modelling Language,* GDML. We are not interested in promoting it as a solution to various problems, but it is convenient to have a name for it as one discusses it and compares it to other notations.

The GDML is based on two underlying language levels that are specified in other lecture notes in our course materials. The lowermost level is the notation of *Knowledge Representation Expressions* (KRE) which shall be understood as an alternative to S-expressions or to the generic XML-like (SGML-based) notation. The KRE notation is similar to S-expressions in the sense that it uses recursively nested, parenthesized expressions where the elements are symbols, strings or numbers, but one difference is that it uses several types of parentheses and brackets, namely, all those that are available on the standard computer keyboard. This makes the notation more readable - the use of just one single kind of parentheses, like in S-expressions and in XML expressions, is often tiring for the eye.

The use of several kinds of brackets allows KRE to write sets and sequences in the standard way from the point of view of mathematics, that is, using

curly brackets for sets and angle brackets for sequences, whereas ordinary, round parentheses are used for terms and composite entities.

Just like there are a number of languages that use the S-expression syntactic style, so there are several languages that use the KR-expression syntactic style. The first of these is the *Common Expression Language,* CEL, which is analogous to Lisp and to KIF: The evaluation of CEL expressions is defined; CEL therefore defines a few 'control' operations such as conditional expressions, and it introduces a number of functions and predicates in particular for operating on scalars and on recursively formed sequences. Besides predicates it also defines the standard propositional connectives and universal and existential quantifiers.

CEL is similar to standard notation in logic and mathematics, and different from Lisp, insofar as it makes a syntactic distinction between variables and entities, and it does not have any counterpart of the quote operator in Lisp. (It does allow Quine quotes on formulas but this is a more specialized facility). Variables are written as symbols where the first character is a point whereas entities are written as symbols where this is not the case, and obeying a few other special-character restrictions. Variables evaluate to whatever they are bound to, and symbols evaluate to themselves, in the case of computational evaluation as described in Section 1.2 of Chapter 1.

The GDML language, which is described in the present lecture note and which is being evolved in the direction of a fully precise definition, is obtained from CEL by adding a number of predicates and functions and characterizing them using axioms. (Previously defined predicates and function in CEL are retained, of course.) The present Appendix contains the first steps in that direction.

Brief Summary of KRE and CEL Notation

The notation that is used here is described in detail in other documents in the present collection, ([5]) but the following is a brief introduction in particular concerning formulas in predicate calculus, which is what is used in the present lecture note.

All formulas on all levels are enclosed by parentheses or brackets. Functions and predicates are written after the opening parenthesis or bracket, rather than before it. The notation differs from S-expressions in that several kinds of brackets are used, as follows:

```
<a b c d>              A sequence
{a b c d}              A set
[r a b :t1 v1 :t2 v2]  A record
[p a b]                A predicate with its arguments
(f a b c)              A function with its arguments
(and [p a b][q c])     A use of a propositional connective
.x                     A variable
a                      A constant symbol
[all .x [q .x]]        A quantified expression
[-p a b]               An abbreviation for (not [p a b])
```

[5] http://www.ida.liu.se/ext/kraic/

The standard propositional connectives are called `not, and, or, imp, eqv.` Of these, `and` and `or` may take an arbitrary number of arguments. The existential quantifier is written as `exist.` Records can have both *arguments* (without tags) and *parameters* (with tags). The number of arguments is fixed for each record operator. If the operator is a predicate then the record is used as a literal, otherwise it is a composite expression that can occur as an argument of a function or predicate, or as an element in a set or sequence. Relationship expressions that are introduced and used in the present lecture note is an example of this.

If one should be annoyed by this dual use of square brackets then it is suggested to prefix predicates with a plus sign, so that

```
[+p a b]     means [p a b]
[-p a b]     means (not [p a b])
```

If these abbreviations are used consistently then a bracketed expression (technically: a KRE record) that begins with a plus or minus sign is a positive or negative literal, one that begins with the quantifier `all` or `exist` is also a proposition, and all others are terms from the point of view of logic.

As a mnemonic convention, function symbols that end with a colon character are used for entity-valued functions that are interpreted as Herbrand functions, like in Prolog. The values produced by such functions are called *composite entities.* Features in GDML are examples of composite entities.

Predicates and Functions for Entities

```
[hastype .e .g]
```

This predicate specifies that the entity `.e` has the type `.g.` Notice that the type is again an entity that has a type, recursively.

Predicates and Functions for Relationships

```
[designates .p .r]
```

This predicate specifies that the entity `.p` is a reification of the relationshp `.r`

```
[hasverb .p .v]
```

This predicate specifies that there exists a relationship `.r` that is formed using the operator `.v` and that satisfies `[designates .p .r]`

```
[param .p .tag .e]
```

This predicate specifies that there exists a relationship `.r` that satisifes `[designates .p .r]` and that contains a parameter with the tag `.tag` and the value `.e`

Predicates and Functions for Features, Actions and Change

`[H .t .f .v]`

This predicate specifies that the feature `.f` has the value `.v` at the timepoint `.t`

`[D .s .t .a]`

This predicate specifies that a process that is an instance of the action `.a` starts at timepoint `.s` and ends at timepoint `.t`

`[created-at .e .t]`

Expresses that the designation of the entity `.e` started to exist at time `.t`

`[destroyed-at .e .t]`

Expresses that the designation of the entity `.e` stopped existing at time `.t`

`(prec .s)`

For discrete time: the timepoint that precedes the one given as argument. This function can be used for both linear and forward-branching time, including the time domain of situation calculus.

`(succ .s .a)`

For the time domain of situation calculus: the successor of situation `.s` that is obtained by performing the action `.a` there.