CASL

Cognitive Autonomous Systems Laboratory **Leonardo**
Department of Computer and Information Science
Linköping University, Linköping, Sweden

Erik Sandewall

Recent Work and Current State of Applications and Ontology in Leonardo

January, 2007

Introduction

The Leonardo Computation System project ([1]) includes work on a novel software architecture and on languages for markup and for knowledge representation, but it also includes a number of experimental applications. The purpose of these is not primarily to provide services for a user community, but (1) to obtain experience with how well the system kernel and platform are suited as the basis for building applied systems, and (2) to obtain evidence for or against the main research hypothesis in the Leonardo project, to the effect that the entire system can be built with little or no duplication of similar concepts and facilities. For these purposes it is sufficient if the applications are in daily use by their own developer (for office applications) or that they can be used for creating convincing and robust demonstrations (for robotic applications). They do not need to have the polish, completeness, documentation and reliability that is needed if the software is to serve a user community.

The research methodology that has been adopted for the project calls for a regular tracking of emerging designs and design changes, in order to document as far as possible the original reasons for various aspects of the design. This means, in particular, that the character and current state of the experimental applications is to be documented from time to time, in such a way that the interested reader of later articles from the project shall be able to go back to the underlying, detailed development and understand these design choices and design changes. The same principle applies for the kernel and platform, that is, the common parts of the system on which the applications are built.

The present memo is one such periodic report, to be accompanied by more detailed reports for some of the experimental applications. It will focus on three particular aspects of the Leonardo system by the end of January, 2007:

- The list of knowledgeblocks representing the kernel and platform, and those representing experimental applications, with a brief explanation of the purpose of each of them and the level of completion and of use.
- An account of the present state of the Leonardo ontology, and of the changes in the ontology since the previously documented design.
- An inventory of how some of the structuring methods that are offered by Leonardo but not by other comparable systems, have actually been used in the experimental applications.

The reason for focussing on these two aspects is that they represent the two major structuring principles, or "backbones" organizing the Leonardo system. The ontology contains entities for the types that are used in the applications. The focus on this information is therefore natural for an overview report.

It should be clear from the above that the present document is *not intended as a research article*, but as background material that can be cited and used as reference in research articles. We also see a possible second use as course material in a course about knowledge representation and ontologies, since

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

in particular the second part of the memo describes a process of organic growth in an ontology, and since it discusses some nontrivial representation problems.

The report describes a number of *design faults* in the ontological structure of the system as it exists today. It is by intention that we first describe the current system with those faults and then correct them; several of them are going to be corrected just after the report has been completed. We do *not* wish to first correct significant faults and then describe the polished result, since that would fail both the development-tracking purpose and the possible pedagogical purpose of this report.

However, a number of trivial faults were corrected in the course of preparing and writing the present report, since they were considered irrelevant for the intended purpose with the report.

The baseline for the present memo is the January, 2007 version of a previous memo "The Leonardo Computation System" ([2]). It will be referred to here as *the baseline memo.* Although the baseline memo was finished and posted shortly before this one, its section concerning the ontology was actually written in August, 2006 and is not up to date. The present memo describes the current state of the ontology and discusses the reasons for the changes between August, 2006 and January, 2007.

The Knowledgeblocks

The baseline memo identified the following knowledgeblocks as constituting the kernel of the Leonardo system:

`core-kb`	`The most central part`
`chronos-kb`	`Management of the calendarial time axis`
`config-kb`	`Creation and modification of system configurations`
`syshist-kb`	`Version management for all information in the system`

This kernel is intended to be kept as constant as possible. The baseline memo also identified the following knowledgeblocks as constituting the system's *platform*:

`channels-kb`	`Channels for message-passing between Leonardo individuals`
`docu-kb`	`Tools for autodocumentation`
`execeval-kb`	`An executive for actions and plans and for delegation of actions between individuals`
`hacker-kb`	`A framework for novice users for getting started with the system`

Notice that an "individual" in Leonardo is an instance of the system that is represented persistently as a collection of files in the directory system of a host computer, and that may contain its own copies of both programs and data (to the extent that that distinction makes any sense).

[2] `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/002/`

These knowledgeblocks are still there in the current system. `channels-kb` and `execeval-kb` are in active use for experiment and demo purposes. The two others, `docu-kb` and `hacker-kb` are used sparingly. The `docu-kb` facility has not yet reached the completeness where it can become the standard autodocumentation tool for the system.

The following is a list of all other knowledgeblocks that are in the system ([3]) at the time of writing, i.e. those that have not already been identified as part of the platform.

proposition-kb	Representation of formulas in first-order logic in Leonardo
text-kb	A text formatter for an in-house text markup language, ADL
compound-kb	Definitions of compounds (explained below)
debug-kb	A small library of operations for debugging support
import-kb	Software tools for importing information from foreign formats
crossix-kb	A simple facility for generating overviews and cross-index tables, in particular for the ontology
madman-kb	Management of documents and manuscripts
caisor-kb	Bibliographic metadata as used by madman, both for bibliographies (a la Bibtex) and for the user's own articles, reports, etc
caisorweb-kb	Like caisor-kb but for webpage and websites
actors-kb	Address information (email, etc.) for persons, in particular those occurring as authors in caisor bibliographies
pub-kb	Domain information for publications, e.g. names of journals and publishers
messaging-kb	Email management
phpcore-kb	Software support for development of PHP programs
phpdata-kb	Copies of entityfiles from actors-kb and pub-kb that are processed using the PHP implementation of a Leonardo subset
agent-kb	Implementation of software agents based on execeval-kb (which is part of the platform)
upatrol-kb	A test example for agent-kb, for simulated intelligent UAV's

[3]The phrase "the system" refers here and in the sequel to the set of knowledgeblocks in the individual called `lar-1`, which is a kind of root individual. Other individuals may have imported, or may share some of those knowledgeblocks, but `lar-1` is comprehensive, with the unimportant exception that larger volumes of application data are sometimes only located in offspring individuals and not in `lar-1` itself.

```
tdda23-kb        Material for course TDDA 23 on Lisp and AI
                   in Linkoeping

devel-kb         A knowledgeblock containing "homeless"
                   entityfiles that arise in the course of
                   ad-hoc implementation of various services
vtest-kb         Some arbitrary test examples
```

The entities that are defined in an individual always include the name for the individual itself. This self-naming entity has an attribute that is called `has-knowledgeblocks`, containing the set of names for the knowledgeblocks that are included in that individual. The list of knowledgeblocks shown above is taken from that attribute for the entity for `lar-1`.

The first six of these knowledgeblocks might be considered for inclusion in the platform, according to their topics. However, `proposition-kb` is in a too early state of development for that to happen yet; when it becomes more complete it will be natural to include it in the platform. On the other hand, `text-kb` is in a stable state since several years (it was written before Leonardo and was ported into the present Leonardo system) but it is due to be replaced by an implementation of a new markup language. Its place in the platform is therefore questionable.

The knowledgeblock `compound-kb` contains a facility for systematic definition, naming, and location of a group of entityfiles. The need for this arose in the context of administrating research articles and manuscripts in `madman-kb`, where these documents are characterized by the name of the series (for example, `caisor` reports), year, and serial number within the year. Each document is associated with relatively much information, including structured data for each of several versions of a document as well as full-text data, for example for the abstract. Each document is therefore represented as several Leonardo entities, in particular, one for each of its versions. Most of the operations involving these entities are done on one document at a time, namely, when the document is edited and formatted. For these reasons it is natural to have a separate file, in the sense of the OS file system, for the information pertaining to each one of the documents, rather than one single file for all of them. This is at odds with the view of entityfiles in the core of Leonardo where each entityfile has a mnemonic name and there is an index that specifies the location for each entityfile. The `compound-kb` knowledgeblock provides a systematic way of handling a group of entityfiles that are defined by a combination of several names, for example, by name of series, year, and serial number.

In principle, this is a general-purpose facility that ought to qualify for the platform. However, other applications have introduced other, more or less ad-hoc conventions for organizing entityfiles and knowledgeblocks. The intention is to make a retrospective on these various arrangements and to look for generally applicable solutions. Meanwhile the `compound-kb` facility is not included in the platform.

The three following knowledgeblocks (`debug-kb, import-kb, crossix-kb`) change too rapidly to be considered as platform facilities. They are actually collections of practical tools that are programmed or reprogrammed just when they are needed. It is therefore not reasonable to include them in a "platform", but one might consider introducing the term of a "workbench" for material that is presently the instrument and/or the object of active work. Some of the previously mentioned blocks, such as `proposition-kb` would also fit on or in a workbench.

The following knowledgeblocks in the list, from `madman-kb` and onwards, do have the character of applications. In some cases there is clear separation between a knowledgeblock for a general facility (`madman-kb`) and knowledgeblocks containing information in a particular usage (`caisor-kb, caisorweb-kb`). In other situations, and in particular as long as the application remains small, general and application-specific information may be kept together in one single knowledgeblock (`compound-kb`, for example).

Some figures: These knowledgeblocks comprise altogether around 190 entityfiles and 3.250 entities. Loading all of them from their textual representation in LDX takes around 30 seconds on a conventional laptop, but this is really only done for the purpose of cross-indexing or other cataloging; in daily use only a small part of the entityfiles are needed, so that the loading process is faster. The operation of loading the core Leonardo system takes place before the library of entityfiles is loaded, and it takes two or three seconds.

Character of use: `madman-kb` and its user data in `caisor-kb, caisorweb-kb,` and `actors-kb` is in daily use by the present author for preparing documents, such as the present one, as well as for amending the caisor website (`http://www.ida.liu.se/ext/caisor/`) which also contains the Leonardo website as a substructure. In addition, `actors-kb` also contains a contacts register (containing addresses, phone numbers, etc) whose contents have recently been imported from the register in a Palm-Pilot; this is just beginning to be used. (The contact entities are not included in the entity-count above). The email management in `messaging-kb` is in a half-way stage of implementation: it can download email message headers from a mail server and organize their information in terms of entities, but the implementation of email sending and of a GUI remains to be done.

The `PHP` implementation and `PHP` support were done in order to gain experience with how well `PHP` is suited for Leonardo data structures, and vice versa. They contain a web-based GUI for access to bibliographic metadata and domain data which works correctly but which is not in actual use. The conclusion from the experiment was that it would be better to use `Python` for this purpose. (An implementation of LDX parsing and printing in `Python` exists as well; it is not a complete system and it is therefore not in actual use).

The `tdda23-kb` knowledgebase for the undergraduate course contains services for capturing session logs; it is in regular use each time the course is given. The agent-oriented knowledgeblocks, finally, which include `agent-kb` and `upatrol-kb` as well as their basis in the platform knowledgeblock `execeval-kb`, are used for tests and demonstration but do not yet constitute an application. They represent the first steps towards porting the WITAS Robotic Dialog Environment (RDE) into Leonardo.

In general, both the kernel, the platform, and the experimental applications in Leonardo have a workbench-oriented character. They are designed for hands-on work on collections of information, where pointwise editing alternates with wholesale operations across small or large sets of entities, as opposed to, for example, search and retrieval in large knowledgebases.

The emphasis on applications for *doing things* has consequences for the type structure or ontology, as we shall see in the next section: the design of the actually occurring ontology in the system has been influenced not only by the intrinsic character of objects in the application domains, but also by pragmatic needs in the operations for doing things to information sets. The question whether this is desirable, undesirable, or just a matter of fact will be saved for later discussions.

The Ontology

Although the experimental applications have a fairly varied pattern of usage, most of them have anyway arrived so far that they provide relevant contributions to the evolution of the Leonardo ontology. This is our second topic.

In ontology development there is a balance between *comprehensive* and *demand-driven* approaches. In the comprehensive approach one tries to create a systematic ontology once and for all, often with a strong component of philosophical considerations. In the demand-driven approach one starts with a simple kernel ontology, and one then proceeds using applications each of which is allowed to contribute extensions to the ontology at hand. In Leonardo we use the latter approach. The reasons for this is that (1) comprehensive ontologies tend to be large and complex; (2) many practical applications do not need very large ontologies; (3) it is doubtful whether a previously designed ontology, even a really large one, will actually serve the needs of an additional application anyway. At the same time we admit of course that there are also arguments in favor of the comprehensive approach, and the present activity should be seen simply as an experiment with using the demand-driven approach for ontology development.

The baseline report contains a description of the original ontology as it was perceived in August, 2006, with around 35 entities in the ontology kernel. Almost six months later the ontology kernel has grown to 60 entities, and the total number of entities in the ontology of `lar-1` is nearly 200. This is still of course a small number compared to comprehensive approaches.

Appendix 1 contains a structured list of the types in this total ontology, organized hierarchically according to the subsumption relation. The list contains two groups, where the first group consists of types that are subsumed by the type `metatype`, and the second (main) group consists of types that are instances of the metatypes in the first group. The relation between type membership and subsumption is explained in the baseline memo, and has not changed. In the large group, the knowledgeblock where the type is defined is indicated in square brackets. All the metatypes in the first group are in `core-kb`.

The Metatype Level

Although the top-level structure is largely unchanged compared to the baseline, there has been one change which may have a certain interest as an example problem. The `metatype` group looks as follows at present:

```
metatype [core-kb]
|--toptype [core-kb]
```

```
|--thingtype [core-kb]
|--qualitytype [core-kb]
|--spacetime-type [core-kb]
|--descriptor-type [core-kb]
|--aggregate-type [core-kb]
```

The graphics means that the entity `metatype` subsumes all of `toptype`, `thingtype`, and so forth. This structure is like in the baseline memo, except that the type `aggregate-type` has been added. This occurred because in several of the applications there arose the need to set up *lists of* entities: lists of persons on a mailing-list, lists of topics that are addressed in a report, and so on. It was convenient to let each list have a name of its own, and to attach various attributes to it besides the sequence of elements in the list. This concept is generic since lists can contain tangible objects, intangible objects, dates, places, and so on, and we ended up making a separate metatype for them.

The Type Level

Proceeding then to the next layer down, in general there is one most general type for each metatype. The most general type in the metatype `qualitytype` is `quality`. All those types that are subsumed by `quality`, for example `color`, also have the type `qualitytype`. Each most general type is the root of a tree, all nodes in which have the same `type`. There are two exceptions to this:

- For the metatype `spacetime-type` there are two most general types, namely `spatial-entity` and `temporal-entity`. This was already in the baseline report.
- With a change relative to the baseline report, all most general types in the sense just described are subsumed by yet another type, called `entity`. The type of `entity` is `metatype`, which then subsumes types such as `thingtype`, `qualitytype`, and so forth. This is shown in figure 1, which can be compared to figure 1 in the baseline report.

The introduction of `entity` in this role does not have any consequences for the modelling of various applications, but it simplified algorithms for analyzing and checking the entire ontology structure. This was evident when we used the routines in `crossix-kb` for cleaning up and checking the ontology that had evolved gradually.

The structure for the plain type (not metatype) level consists of subtrees whose roots are as follows: `thing`, `quality`, `spatial-entity`, `temporal-entity`, `leoslot`, and `aggregate-entity`. We shall address them in turn, although we save `thing` until last since it is by far the largest group.

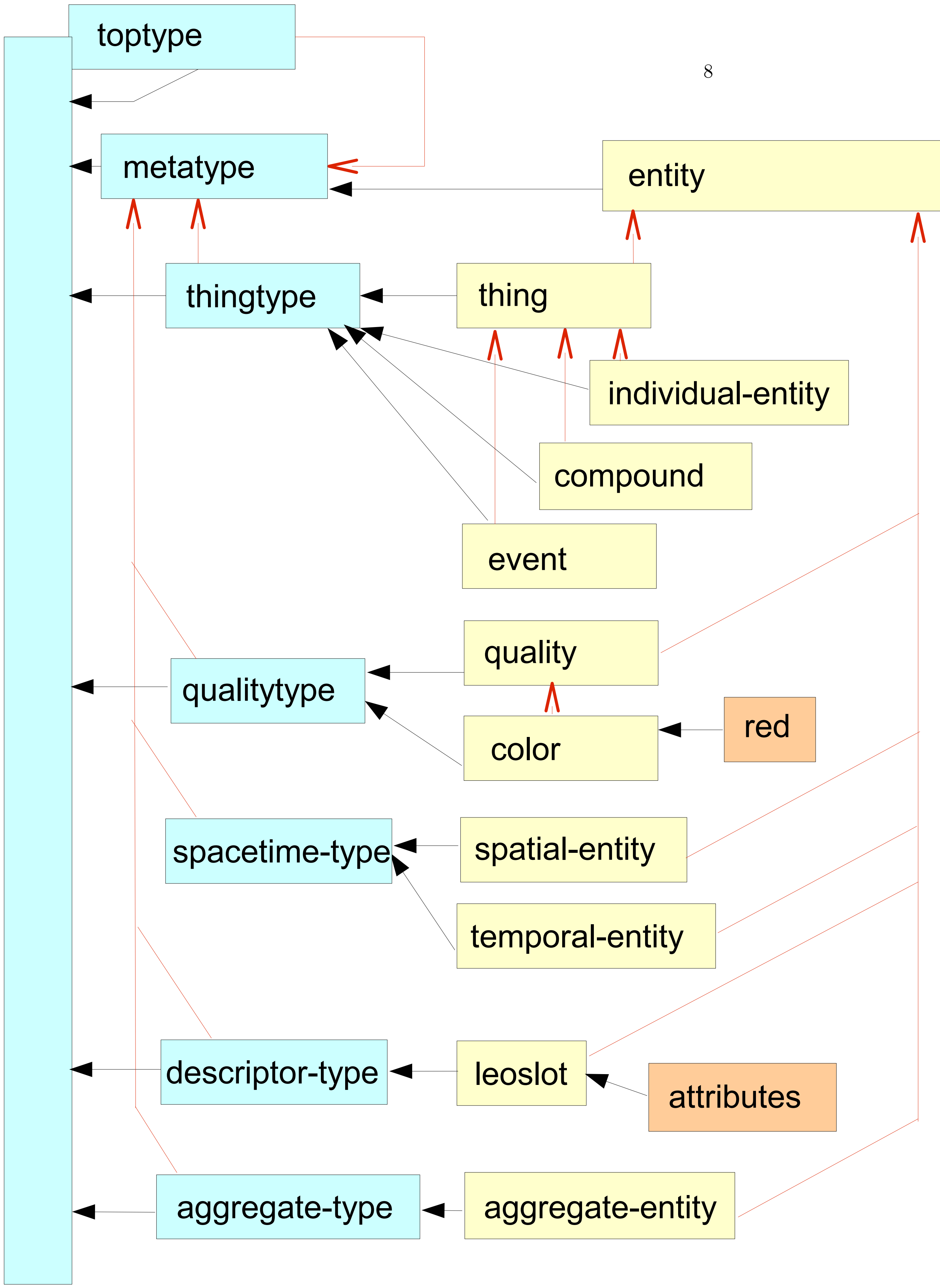

Figure 1: Top level of Leonardo ontology

The Ontological Substructure under `quality`

This contains

```
|--quality [core-kb]
|--|--color [core-kb]
```

This group was introduced in the initial ontology design and from general, abstract considerations. The entity `color` was put into the ontology in order to have an instance and an example there, but until now there has not been any use at all of this structure in the applications.

Does this mean that this substructure is unnecessary, or does it mean that we have not yet arrived to the point in the applications where it will be needed? A priori one would expect instances of `quality` to occur as attribute-values and as arguments to predicates such as `Holds`. As long as these attribute-values are just tokens there is no particular need to classify them by type, but the need does arise in at least two contexts: (a) when structure specifications are introduced for attribute values; compare the discussion of this topic in section 4 of the LDX representation language report ([4]); (b) if attributes need to be assigned to these properties, for example (in a robotic context) recognition criteria for a particular quality. The first of these will certainly arise; the second one is also likely to occur. It is therefore natural to retain this structure in the ontology.

The Ontological Substructure under `spatial-entity`

This contains

```
|--spatial-entity [core-kb]
|--|--city [actors-kb]
|--|--country [actors-kb]
|--|--addressable-location [actors-kb]
|--|--|--workplace [actors-kb]
|--|--|--residence [actors-kb]
|--|--|--summerhouse [actors-kb]
```

The types `city` and `country` were introduced in `actors-kb` since address information for persons and for organizations (publishers, for example) contain these elements, and it was felt natural to structure it as entities and not merely represent this information as dumb strings.

The type for `addressable-location` is an abstraction of the three types that it subsumes. This was introduced for the contacts register, where it was desired to introduce entities for each one of several “addresses” that a person or a family may have, and where each “address” can have attributes for street address, landline telephone number, instructions for getting there, etc. There is a slightly philosophical question whether the “summerhouse”, for example, ought to be considered as a tangible object namely the building, or whether it ought to be considered as a kind of geographical object. In this representation we take the latter position.

[4]`http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/001/`

The Ontological Substructure under `temporal-entity`

This contains

```
|--temporal-entity [core-kb]
|--|--date [core-kb]
|--|--|--date-details [core-kb]
|--|--leoyear [core-kb]
|--|--memorypoint [core-kb]
|--|--milestone [agent-kb]
|--|--bucyear [caisor-kb]
|--|--caisoryear [caisor-kb]
```

The type `bucyear` has members for each year during the last few decades; each such entity is used for keeping track of articles and reports that have been published during a particular year. The type `caisoryear` is similar. The instances of `milestone` are entities that represent points to be achieved during the execution of a plan. The other types are used for internal purposes in the Leonardo system itself, in particular for the version manager and for managing session logs.

The concept of a *date* is actually represented both using records and using entities. The record representation is as e.g. `[date: 2006 11 21]` with the obvious meaning. It is used in attribute-values but does not allow attaching any information to the date in question, and for that purpose one must use entities.

The ontology contains two entities called `date` and `date-details`, for the following reasons. The `date-details` type is application specific and an instance of this type carries a particular kind of information about a particular date. The `date` type was introduced because it was felt that there should be some ontological entity for the very concept of a calendarial date, but is is only used as a subsumer for `date-details` at this point and does not have any instances of its own.

Instances of `date-details` are usually composite entities formed like in `(date-details: 2006-10-01)`. Every such entity is a catalog containing entities of the type `chronicle` that occurred during that day. A chronicle may simply be one activation, or "run" of a Leonardo individual. It consists of a sequence of `event` entities, for representing events that have occurred during the activation. These events may be grouped using `episode` entities. An episode names a set of events within a chronicle that one may wish to refer to for some particular reason, for example, because they demonstrate a capability of a human-computer dialog system, or because they demonstrate the presence of a bug.

Each `date-details` entity obtains its own entityfile, containing itself and its chronicles and episodes. In that respect it could be considered as subsumed by `entityfile`.

The chronicles in an entity such as `(date-details: 2006-10-01)` obtain names such as `Chronicle-2006-10-01-002` which are generated automatically using a numbering scheme. In retrospect it might have been better to consider them as composite entities as well, for example
`(chronicle: (date-details: 2006-10-01) 2)`.

The type `chronicle` is subsumed by `activity` and therefore by `event` and by `thing` in the ontology. This structure is disjoint from `temporal-entity` which subsumes `date` and `date-details`. This may have to be reconsidered at some point.

The representation for *years* is analogous in the sense that there are several year-types at this point, but for them there is no common, generic concept, and only three different "year" concepts that are specialized for particular purposes. Instances of the `bucyear` and `caisoryear` types are used for lists of publications during specific years, in the `madman-kb` application. Two different types were introduced because one is used for the direct information about publications during specific years, and the other one is used for automatically computed attributes that are used for converting between internally and externally used names for the same publication. Instances of `leoyear`, on the other hand, are only used in the context of version management, for keeping track of the successive numbering of "synchronization" timepoints within the year.

At present the instances of `bucyear` are formed as a symbol like `|2006|`, the instances of `leoyear` are formed like `year-2006`, and the instances of `caisoryear` are formed like in `(caisoryear: 2006)`. In retrospect it would be better to use composite entities in all cases, to merge `bucyear` and `caisoryear` into one type, and to introduce a generic type for years in general.

The Ontological Substructure under `leoslot`

This contains

```
|--leoslot [core-kb]
|--|--leoprofile [core-kb]
```

The original idea was that names for attributes and other constructs that are used for technical reasons in Leonardo knowledgeblocks should be declared as instances of `leoslot` or types that it subsumes. In practice this has not happened except for a few odd cases; the present system has not given any good reasons for making such a declaration. It will be needed however, when structure specifications are introduced for attribute values (compare above).

This group has obvious overlap with, and borderline issues with the groups of both `computation-entity` and `model-entity`. One may consider the possibility of treating this as another group under the umbrella of the type of `intangible-entity`, instead of a top-level structure of its own.

The Ontological Substructure under `aggregate-entity`

This contains

```
|--aggregate-entity [core-kb]
|--|--ef-list [actors-kb]
|--|--personlist [actors-kb]
|--|--topiclist [caisor-kb]
|--|--propogroup [proposition-kb]
```

These represent types for lists of entityfiles, lists of persons, lists of topics (of research articles, for example), and lists of proposition-groups, respectively. Proposition-groups are used for lists of propositions in logic.

These four groups had been introduced ad hoc as the need arose in different applications, and the introduction of a common structure for them was done as an afterthought. This group was not present in the baseline report, as has already been explained.

In retrospect one would be tempted to introduce a symbolic function `listof:` that has a type as an argument and a type as a value, so that one could write `(listof: entityfile)` instead of the symbol `ef-list`. Notice, however, that the instances of the types mentioned above do not *merely* represent sequences of things; they may also carry other information in particular with respect to intended use. For example, each mailing-list may carry information about who manages it, about whether it is open or closed, on which server it operates, and so on. In such a case, the type for mailinglists should be subsumed by `(listof: entityfile)`, and therefore also by `aggregate-entity`, but it should not *be* `(listof: entityfile)`.

The `aggregate-entity` type should be open for other kinds of aggregates, besides lists of things. In fact, an ontology would be a good example of an aggregate-entity, with its type structure and subsumption structure.

The Ontological Substructure under `thing`

This substructure consists of exactly 100 entities in the ontology at the time of writing, which means that it occupies more than half of the total set of entities. We refer to the full list in the appendix and do not include the entire list here, but only some excerpts.

Consider first the following substructure, containing three groups that belong under `intangible-entity`:

```
|--|--|--|--cognition-entity [core-kb]
|--|--|--|--|--method [execeval-kb]
|--|--|--|--communication-entity [core-kb]
|--|--|--|--|--output-phrase [execeval-kb]
|--|--|--|--|--leomail-object [messaging-kb]
|--|--|--|--convention-entity [core-kb]
|--|--|--|--|--language [actors-kb]
```

Among these, `cognition-entity` and `communication-entity` appeared in the baseline ontology, but `convention-ontology` was introduced because there was a concrete need to have a type for `language` with instances such as `english`, `french`, `chinese`, etc. representing those respective languages, which can be used e.g. for specifying the language in which a particular document is written. None of the previous major kinds of `intangible-entity` seemed to be able to subsume `language` appropriately, which is why `convention-entity` was introduced, with a hope that it could also be used for other kinds of behavior patterns.

The type called `method` is used in `execeval-kb` for a method that can be used for achieving a particular goal, in the sense of goal-directed behavior.

It is clear that these structures are very sparse, in the sense that there are so many other kinds of cognition entities and communication entities besides the occasional ones shown above.

The baseline design included a general type for `abstract-entity`, which was intended to subsume concepts such as `triangle` (representing the triangular shape, not the music instrument). Since then an additional group was added for `model-entity` so the table looks as follows:

```
|--|--|--|--model-entity [core-kb]
|--|--|--|--|--grammar-item [core-kb]
|--|--|--|--|--predicate [proposition-kb]
|--|--|--|--abstract-entity [core-kb]
```

The idea was that a `model-entity` would be a concept that is introduced in order to be used in "models" of the world or of an application, if the construct does not already fit naturally into some other part of the structure. Philosophically there seems to be some difference between `abstract-entity` and `model-entity` but the borderline is not sharp and this part of the design may need to be revised again.

The current ontology contains a quite large section led by `info-entity` which is a new one that did not appear in the baseline. It was introduced in order to accomodate bibliographic items representing books, journal articles, technical reports, and so forth. There may be a philosophical issue concerning the attempted distinction between `communication-entity` for such things as letters and emails and, on the other hand, `info-entity` for items without an explicit addressee. There are obvious borderline cases: what about an announcement in an airport public address system that begins with "attention all passengers" but continues with information about a particular departure? In spite of the overlap we hypothesize that the distinction is a useful one in practice.

Comparing the groups subsumed by `intangible-entity` in the baseline report and here, three groups have been added, namely `cognition-entity`, `model-entity`, and `info-entity`, and none has been removed.

With respect to the groups subsumed by `tangible-entity`, the following structure contains certain interesting problems:

```
|--|--|--|--person-name [actors-kb]
|--|--|--|--|--contact-name [actors-kb]
|--|--|--|--|--authors [compound-kb]
|--|--|--|--|--email-alias [messaging-kb]
|--|--|--|--|--author [pub-kb]
|--|--|--|--|--author-here [pub-kb]
```

Here, a `person-name` is intended as an entity that has attributes specifying the email address, cellphone number, and maybe also street address and home phone number for a particular person. (Alternatively, the person may have an attribute specifying an entity for his/her family, which in turn has an attribute for the home(s) that the family uses, and these again may be associated with street address etc). Now, should this type be called `person` or something like `person-name`? The philosophical reason for this possibly strange question is that usually when we represent physical objects of various kinds in the system, we represent properties that are intrinsic to them, but things such as name and email address are attributes that have been *attached to* the person in question. If it is considered worthwhile to make this distinction, then things such as bodyweight, place of birth, and references to parents and children ought to be attributes for the `person`, whereas given name, family name, and email address are attributes for the `person-name`, or whatever we wish to call this façade of the actual person.

Another representation problem: some people have more than one email address, and each of these may have to be associated with information about when and for what it is to be used, for example. This means that there is a need for several entities in such cases. However, introducing one entity for each email address of every person may be to overdo it, since for most people the system only needs to list a single email address, which can then be an attribute of the person's entity in the system. The solution that was chosen in `actors-kb` was to introduce a separate entity, of type `email-alias`, for each email address besides the first one for a given person. Each instance of `person-name` and of `email-alias` can have an `email` attribute, containing the actual email address as a string. There is an optional attribute that assigns an entity of type `email-alias` to an entity of type `person-name`. This is a workable practical solution, but what kind of thing is the email-alias for a particular person? Are aliases in general a type, subsuming alias types for email addresses, for the car driven by a particular person, and so forth? Or, is an email-alias merely a variant of the type for `person-name` but with a more restricted set of attributes?

The types for `author`, `author-here`, and `authors` are all used for bibliographic metadata, and are different types for the same kinds of things. They refer to a person that is an author, not to "the author" of a particular article, so if the same person has written several works it is still one single entity for the purpose of these types. (Notice also the overlap with the type of `person-name` in the contacts register). The type `author-here` is subsumed by `author` and is used for those persons that have at some time been a co-author of the user of the system at hand, in which case additional information may have to be represented. The type called `authors` is used for a single author (not for a group of authors, as the name might suggest), and was introduced as bibliographic metadata were imported from two different sources, both of which sometimes contributed information about the same author. It was convenient to have two distinct entities for the same author, one for each source, for the work on importing, adjusting, and correcting the information from the two sources. Once information has been imported, one should be able to use the facilities in the system to merge the information to a more reasonable representation.

This example illustrates the dynamic character of types in a system like Leonardo, and is interesting in combination with the case of multiple representations of the concept of a year which was mentioned above. In general, type distinctions arise not only because of distinctions in the knowledge that is being represented, but also because of pragmatic needs when working with the information, and because of rapid design decisions when facing a particular application needs. Import of information from external knowledge sources sometimes requires active manipulation, and types may need to be introduced ad hoc in order to facilitate that work. At the same time, one of the major reasons for having a systemwide ontology in the first place is to be able to relate information that is obtained from different sources. It is therefore a major requirement on a system of this kind that it shall support the *homogenization* process where information that has arrived or been set up in the form of different types can be transformed and integrated, so that it obtains a common type.

Some items in the structure can be considered as "noise" and should be eliminated at the earliest occasion, in particular the dual occurrence of some type names in both capital and small letters. This is a legacy problem, due to migration of some data from an earlier implementation in an upper-case-only variant of Lisp. Some other items are obviously incorrect, such as the inclusion of `goal` and `goal-instance` as subsumees of the same subsumer. This does not hurt for the moment but will have to be straightened out.

Structures of Particular Interest

Symbolic functions

One of the characteristic facilities in the Leonardo Data Expression language (LDX) is the possibility to form *composite entities* using *symbolic functions*. The following is an inventory of how this facility has actually been used in the Leonardo system as it exists at the time of writing this report. We present the symbolic functions in groups according to the knowledgeblocks where they are defined.

```
location: [core-kb]
```

An entity `(location ef)` where `ef` is the name of an entityfile, is used for carrying information about the location of that entityfile in the directory structure.

```
caisoryear: [caisor-kb]
date-details: [chronos-kb]
```

These symbolic functions were described in the section on temporal entities, above.

```
indiv-loc: [config-kb]
indiv-pode: [config-kb]
```

An entity such as `(indiv-loc: lar-001)` has an attribute specifying the current physical location of the Leonardo individual `lar-001`, for example on a hard-disk unit of a particular computer host, or on a particular detachable USB memory. An entity such as `(indiv-pode: lar-001)` contains port descriptions (therefore `pode`) for that same Leonardo individual. These have the type `indiv-loc-descr` and `indiv-port-descr`, respectively. Entities in these types are maintained in a global place so that they can be shared between several individuals.

```
b: [execeval-kb]
i: [execeval-kb]
episode: [execeval-kb]
simulation: [execeval-kb]
fea: [execeval-kb]
```

If `a` is an entity representing an action and `t` is a timepoint, then the entity `(b: a t)` represents an instance of that action starting at timepoint `t`. The function `i:` is similar to `b:` but refers only to the initiation of the action instance, that is, the activity where it is checked whether it is well-defined and executable.

The function `episode:` is used for setting up multiple episodes within a given chronicle, in the way that was discussed in the section on temporal entities, above.

The functions `simulation:` and `fea:` are explained by the following example. Suppose you wish to simulate cars that are moving in a road system. The entity `(simulation: car)` can represent the type of simulated cars, as distinct from real cars. One of its attributes is `featuremap` which may have the following value, in an overly simple example,

```
{[: velocity 0] [: road 1] [: roadpos 2]}
```

meaning that the "features" or "state variables" `velocity`, `road` (for the identifier of the road that the car is presently driving along), and `roadpos` (for the position along that road) should be in positions with index 0, 1, and 2, respectively in a vector representing the car in the simulation. Let `car24` be one instance of `(simulation: car)`. Then `(fea: roadpos car24)` is an entity representing the road position of `car24`; it may contain information e.g. about how often that state variable is to be updated.

```
remex: [execeval-kb]
achieve: [execeval-kb]
adogoal: [execeval-kb]
ev: [execeval-kb]
get: [execeval-kb]
query: [execeval-kb]
ask: [execeval-kb]
ans: [execeval-kb]
plask: [execeval-kb]
goto: [execeval-kb]
go-shopping: [execeval-kb]
```

The symbolic functions in this group are *action verbs* which are used for forming actions that can occur as arguments of `b:`. The `remex:` operator has a particular meaning: If `i` is a Leonardo individual and `a` is an action, then `(remex: i a)` is the action in the present individual where it delegates to the individual `i` to perform action `a`.

If `g` is a description of a goal, that is, a condition in the world, then `(achieve: g)` represents the action of doing what it requires to achieve that goal. The action `(adogoal: g)` represents the action of adopting the goal, that is, of deciding to try to achieve it.

The verb `ev:` stands for "evaluate": if `f` is a form then `(ev: f)` is the action of evaluating the form and producing its value.

The other action verbs have simple operative definitions and are used for testing purposes. The action verbs `ask:` and `ans:` are used for the actions where one Leonardo individual asks a question to another one, and the latter one answers.

```
fly-to: [upatrol-kb]
mount: [upatrol-kb]
unmount: [upatrol-kb]
```

Composite entities formed using these symbolic functions represent actions in the beginnings of the UAV simulator for the RDE system (Robotic Dialog Environment). The `mount` and `unmount` actions refer to attaching an instrument to the simulated UAV and for detaching it from the UAV.

```
workplace: [actors-kb]
residence: [actors-kb]
summerhouse: [actors-kb]
```

If `f` is an entity representing a family, then `(residence: f)` represents the place where this family lives. This entity has attributes for street address, phone number, etc. The entity `(summerhouse: f)` is analogous. If `p` is a person then `(workplace: p)` represents the place of work for that person, again with attributes for address, phone number for the main telephone exchange, etc.

```
docversion: [caisor-kb]
version: [madman-kb]
```

If `d` is an entity representing an article or report and `n` is a positive integer, then `(version: d n)` represents version number `n` of the article. The difference between `docversion:` and `version:` is trivial and these two should be unified.

```
caisordoc: [madman-kb]
lectnote: [madman-kb]
lecturenote: [compound-kb series-compound]
witas: [madman-kb]
aijd: [madman-kb]
epos: [madman-kb]
epress: [madman-kb]
casl: [madman-kb]
cais: [compound-kb]
etaicontr: [compound-kb]
racnote: [compound-kb]
epcis: [compound-kb]
pury: [compound-kb]
```

The symbolic functions in this group are used for forming document entities. For example, `(casl: 2006 4)` represents report number 4 during 2006 from the CASL research group. The entity `(pury: 1998 114)` represents an article in the publication register of our department, viz. article registered as number 114 during year 1998, and so forth. In a few cases there are accidental duplications, with two functions that denote the same thing, in particular `caisordoc:` and `cais:`.

```
email-alias: [messaging-kb]
```

If `p` represents a person for which two different email addresses need to be represented, then `(email-alias: p)` is an entity representing the person's "second" email address. It has an attribute for the email address itself as a string, and optional other information describing the use of this email address.

Concluding Remarks

Ontological Issues

We have already touched on the fact that Leonardo system design is oriented towards workbench-type or "hands-on" type applications. We also observed that one consequence of this is that the type structure that has evolved during fairly direct implementation of a number of applications as they were needed, is influenced not only by the conceptual structure of the objects in the application domains, but also by the pragmatic and operational requirements in each application and facility.

Furthermore, the type structure or ontological structure in our case is first of all a way of structuring a number of applications, and for this purpose it does not necessarily have to be very comprehensive. It suffices if it provides a reasonable "top level ontology" where contributions from different applications can be inserted as they emerge. This is again different from systems for information retrieval in large knowledgebases that require large ontologies, and where the pragmatic and operational aspects can be expected to be much better separated from the ontology.

However, it is still important, even in our type of applications, that the ontology is capable of scaling up. The one described here includes about 200 types; the methodology and the system should be capable of managing ten times that, and preferably fifty times that. Our present system will continue to be extended under the assumption that it is capable of such growth.

A second, important use of the ontology is for the retroactive analysis and cleaning up of a number of applications using overlapping information categories, after they have first been implemented. The discussion in the present report is one example of such a retroactive analysis. This requires software tools that harvest the collection of all the knowledgeblocks in a Leonardo system, in order to obtain systematic lists of types, of symbolic functions, and so forth. The `crossix-kb` knowledgeblock in Leonardo provides such a facility and was used for preparation of the present report.

The integration of truly ontological considerations with pragmatic ones is natural in the development stage, but in a longer perspective this distinction ought to be made more explicit. It is easy to think of formal devices for doing this. The case of the representation of years, which occurs in the `temporal-entity` structure, suggests that one might use a symbolic function `aspect:` with a convention whereby `year` is the type for years *per se*, `(aspect: year chronos-kb)` is the type for the aspect of years that is used by the knowledgebase `chronos-kb`, and so on. There are also other designs that come to mind. This must be addressed in a systematic manner in due time, but first we must obtain reliable and well documented information about what the actual needs are, and which pragmatic and operational considerations must be taken into account.

Methodological Issues

This report has described a system that was allowed to develop fairly freely during a period of half a year. This has resulted in the organic identification of a number of specific problems each of which has obtained a solution along the way, so the process has helped focus the design of the architecture and the ontology on relevant issues. However, it has also resulted in a number of design faults that must now be reviewed and corrected, and the present candid report has identified several of them.

One may consider whether it would have been better to start with a larger and more thought-out ontology, in particular one that is available in the literature, and whether that would have helped us avoid the kinds of faults that were reported here. For such an analysis one needs to assess the following questions:

1. Would an existing ontology have provided good solutions for the problems mentioned here?
2. How much work is involved in learning to use a large, previously published ontology?
3. How much work is required to correct the faults in the existing system, including both the resulting changes to the program "code" and the update of the existing data sets?
4. Have there been any losses in the development work until now due to the ad hoc nature of some of the design decisions?

We do not have any ready answers to these questions. The first one is a well defined question that could be the topic of a separate study. Question 2 might be the topic of another investigation, and Question 3 can be answered by just doing the job, which needs to be done anyway. For Question 4 the answer will be more speculative. We expect that forthcoming project memos will address some of these questions.

Appendix 1

The following is a listing of all the types in the present `lar-1` individual, which contains 193 entityfiles and 3253 entities excluding a few large collections of uniformly structured entities. The listing is hierarchically ordered for subsumption in the obvious way, so that at the beginning of the second group, `entity` subsumes `thing` which in turn subsumes `individual-entity`, and so forth. Entities that are aligned directly under each other are therefore directly subsumed by the same subsumer.

```
metatype
|--toptype
|--thingtype
|--qualitytype
|--spacetime-type
|--descriptor-type
|--aggregate-type

entity [core-kb]
|--thing [core-kb]
|--|--individual-entity [core-kb]
|--|--|--intangible-entity [core-kb]
|--|--|--|--computation-entity [core-kb]
|--|--|--|--|--globvar [core-kb]
|--|--|--|--|--indivattr [core-kb]
|--|--|--|--|--leo-individual [core-kb]
|--|--|--|--|--location [core-kb]
|--|--|--|--|--os-command [core-kb]
|--|--|--|--|--computer-host [config-kb]
|--|--|--|--|--data-carrier [config-kb]
|--|--|--|--|--indiv-loc-descr [config-kb]
|--|--|--|--|--indiv-port-descr [config-kb]
|--|--|--|--|--agent [agent-kb]
|--|--|--|--|--stringmapping [messaging-kb]
|--|--|--|--|--phpage [phpcore-kb]
|--|--|--|--cognition-entity [core-kb]
|--|--|--|--|--method [execeval-kb]
|--|--|--|--communication-entity [core-kb]
|--|--|--|--|--output-phrase [execeval-kb]
|--|--|--|--|--leomail-object [messaging-kb]
|--|--|--|--convention-entity [core-kb]
|--|--|--|--|--language [actors-kb]
|--|--|--|--leo-entity [core-kb]
|--|--|--|--|--bundle [core-kb]
|--|--|--|--|--entityfile [core-kb]
|--|--|--|--|--|--kb-index [core-kb]
|--|--|--|--|--|--startup-file [core-kb]
|--|--|--|--|--|--simulation-base [execeval-kb]
|--|--|--|--|--section [core-kb]
|--|--|--|--|--leo-residence [config-kb]
|--|--|--|--|--leochannel [config-kb]
|--|--|--|--info-entity [core-kb]
|--|--|--|--|--infosite [caisor-kb]
|--|--|--|--|--hypernote [caisor-kb]
```

```
|--|--|--|--|--WEBNOTE [caisor-kb]
|--|--|--|--|--webnote [caisorweb-kb]
|--|--|--|--|--etai-submission [compound-kb]
|--|--|--|--|--document [compound-kb]
|--|--|--|--|--|--docresource [caisor-kb]
|--|--|--|--|--|--docversion [caisor-kb]
|--|--|--|--|--|--projmemo [caisor-kb]
|--|--|--|--|--|--projmemo-version [caisor-kb]
|--|--|--|--|--|--projreport [caisor-kb]
|--|--|--|--|--publication [pub-kb]
|--|--|--|--|--|--acamemo [caisor-kb]
|--|--|--|--|--|--acamemo-version [caisor-kb]
|--|--|--|--|--|--ep-purwork [compound-kb]
|--|--|--|--|--|--ep-racnote [compound-kb]
|--|--|--|--|--|--ep-article [compound-kb]
|--|--|--|--|--|--book [pub-kb]
|--|--|--|--|--|--|--BOOK [pub-kb]
|--|--|--|--|--|--collection [pub-kb]
|--|--|--|--|--|--incollection [pub-kb]
|--|--|--|--|--|--|--INCOLLECTION [pub-kb]
|--|--|--|--|--|--inconference [pub-kb]
|--|--|--|--|--|--|--INCONFERENCE [pub-kb]
|--|--|--|--|--|--injournal [pub-kb]
|--|--|--|--|--|--|--INJOURNAL [pub-kb]
|--|--|--|--|--|--invit-abstract [pub-kb]
|--|--|--|--|--|--|--INVIT-ABSTRACT [pub-kb]
|--|--|--|--|--|--report [pub-kb]
|--|--|--|--|--|--|--REPORT [pub-kb]
|--|--|--|--|--|--PUBLICATION [pub-kb]
|--|--|--|--model-entity [core-kb]
|--|--|--|--|--grammar-item [core-kb]
|--|--|--|--|--predicate [proposition-kb]
|--|--|--|--abstract-entity [core-kb]
|--|--|--tangible-entity [core-kb]
|--|--|--|--building [core-kb]
|--|--|--|--person-name [actors-kb]
|--|--|--|--|--contact-name [actors-kb]
|--|--|--|--|--authors [compound-kb]
|--|--|--|--|--email-alias [messaging-kb]
|--|--|--|--|--author [pub-kb]
|--|--|--|--|--author-here [pub-kb]
|--|--|--social-entity [core-kb]
|--|--|--|--enterprise [config-kb]
|--|--|--|--|--enterprise-name [actors-kb]
|--|--|--|--|--confseries [pub-kb]
|--|--|--|--|--|--CONFSERIES [pub-kb]
|--|--|--|--|--journal [pub-kb]
|--|--|--|--|--|--JOURNAL [pub-kb]
|--|--|--|--|--publisher [pub-kb]
|--|--|--|--|--|--PUBLISHER [pub-kb]
|--|--|--|--family [actors-kb]
|--|--|--simultype [execeval-kb]
|--|--event [core-kb]
|--|--|--leosession [core-kb]
```

```
|--|--|--activity [core-kb]
|--|--|--|--chronicle [core-kb]
|--|--|--actionverb [execeval-kb]
|--|--|--action [execeval-kb]
|--|--|--|--archivepoint [core-kb]
|--|--|--action-instance [execeval-kb]
|--|--|--goal [execeval-kb]
|--|--|--goal-instance [execeval-kb]
|--|--|--conference [pub-kb]
|--|--compound [compound-kb]
|--quality [core-kb]
|--|--color [core-kb]
|--spatial-entity [core-kb]
|--|--city [actors-kb]
|--|--country [actors-kb]
|--|--addressable-location [actors-kb]
|--|--|--workplace [actors-kb]
|--|--|--residence [actors-kb]
|--|--|--summerhouse [actors-kb]
|--temporal-entity [core-kb]
|--|--date [core-kb]
|--|--|--date-details [core-kb]
|--|--leoyear [core-kb]
|--|--memorypoint [core-kb]
|--|--milestone [agent-kb]
|--|--bucyear [caisor-kb]
|--|--caisoryear [caisor-kb]
|--leoslot [core-kb]
|--|--leoprofile [core-kb]
|--aggregate-entity [core-kb]
|--|--ef-list [actors-kb]
|--|--personlist [actors-kb]
|--|--topiclist [caisor-kb]
|--|--propogroup [proposition-kb]
```