CASL

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

Erik Sandewall

The Leonardo Computation System

This project memo pertains to the development of the Leonardo system. Identified as PM-leonardo-002, it is disseminated through the CAISOR website and has URL `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/002/`

Related information can be obtained via the following www sites:

CAISOR website:	`http://www.ida.liu.se/ext/caisor/`
CASL website:	`http://www.ida.liu.se/ext/casl/`
Leonardo system infosite:	`http://www.ida.liu.se/ext/leonardo/`
The author:	`http://www.ida.liu.se/~erisa/`
Date of manuscript:	2007-01-22

Abstract

The purpose of the research reported here is to explore an alternative way of organizing the general software structure in computers, eliminating the traditional distinctions between operating system, programming language, database system, and several other kinds of software. We observe that there is a lot of costly duplication of concepts and of facilities in the conventional architecture, and believe that most of that duplication can be eliminated if the software is organized differently.

This article describes Leonardo, an experimental software system that has been built in order to explore an alternative design and to try to verify the hypothesis that a much more compact design is possible and that concept duplication can be eliminated or at least greatly reduced. Definite conclusions in those respects can not yet be made, but the indications are positive and the design that has been implemented so far has a number of interesting and unusual features.

The author's present affiliation is:

```
Department of Computer and Information Science
Linköping University
Linköping, Sweden
```

For the author's up-to-date webpage and E-mail coordinates, please refer to the article's URL which is specified on the front page.

1 Introduction

Project Goal and Design Goals

Leonardo is a software project and an experimental software system that integrates capabilities that are usually found in several different software systems:

- in the operating system
- in the programming language and programming environment
- in an intelligent agent system
- in a text formatting system

and others more. I believe that it shall be possible to make a much more concise, efficient, and user-friendly design of the total software system in the conventional (PC-type) computer by integrating capabilities and organizing them in a new way.

The **purpose of the Leonardo project** is to verify or falsify this hypothesis. This is done by designing and implementing an experimental system, by iterating on its design until it satisfies a number of well defined criteria, and by implementing a number of characteritic applications using the Leonardo system as a platform.

The implementation of the experimental system has passed several such iterations, and a reasonably well-working system is in daily use at the time of this writing. The following are the requirements that were specified for that system and that are satisfied by the present implementation. We expect to retain them in future system generations.

The system is of course organized in a modular fashion, where the modules are called *knowledge blocks* and contain both algorithms, data, and intermediate information such as ontologies and rules. There shall be a designated *kernel* consisting of one or a few knowledge blocks that is used as a basis on which other blocks can be built, for the purpose of additional services and for applications. The following were and are **the requirements on the kernel**:

- It shall contain self-describing information and corresponding procedural capabilities whereby it is able to administrate itself, its own structure, and its own updates.
- It shall provide the extension capabilities that make it possible to attach additional knowledge blocks to it and to administrate them in the same way as the kernel administrates itself.
- It shall provide adequate representations for the persistent storage of all contents of the blocks in the kernel, as well as the representations and the computational services for performing computations on the same contents.
- It shall provide capabilities for adaptation, in particular to facilitate moving a system between hosts, and for defining alternative configurations based on different sets of knowledge blocks.

- Although the experimental system will be based on an existing, conventional operating system and ditto programming language, it shall be designed in such a way that it can be ported to the weakest possible, underlying software base.

The last item in these requiements is included because in principle we believe that the services of the operating system and the programming language and system, should just be parts of one integrated computation system. The longer-term goal is therefore that the Leonardo system itself should contain the programming-language and operating-system services.

Furthermore, the facilities in the kernel have been, and will continue to be designed in such a way that they do not merely serve the above-mentioned requirements on the kernel itself; they shall also be general enough to provide a range of applications with similar services.

Above the kernel and below the specific application areas and applications, there shall also be an extensible *platform* consisting of knowledgeblocks that are of general use for a number of applications of widely different character.

Main Hypothesis for the Leonardo Project

The main hypothesis for this project, for which we hope to obtain either strong positive evidence or a clear refutation, is as follows: *It is demonstrably possible to design the kernel and a platform in such a way that (1) repeated implementation of similar tasks is virtually eliminated in the kernel and platform, and (2) the total software structure that is obtained when several applications are built on this platform can also be essentially free from repeated implementations of similar tasks.*

Approach to the Design

The design of the system does not start by defining a programming language, nor by defining a process structure or a virtual instruction set. In Leonardo, the first step in the design is to define an object-oriented information structure that has some points in common with RDF ([1]) and OWL ([2]), although also with significant differences. The notation used for this purpose is called LDX, the Leonardo Data Expression language. It is used for all information in the system, including application data, procedures, ontologies, parameter structures, and whatever corresponds to data declarations in our system.

The element in the LDX structure is called an *entity*, and entities can have *attributes* and *properties*. Attribute values can have structure and are not merely links to other entities; they can be constructed by the formation of sets, sequences, and records, even recursively. Moreover, entities can be composite expressions; they are not merely atoms with mnemonic names. Property values are like long strings and can be used for expressing e.g. a function definition, or a descriptive comment. Because of this expressive power, LDX is best viewed as a knowledge representation language.

[1] `http://www.w3.org/RDF/`
[2] `http://www.w3.org/TR/owl-features/`

We use the term 'entity' rather than 'object' for the elements in LDX since the term 'object' has a connotation of message passing and a fairly restrictive view of class hierarchy, which are not applicable in LDX.

Each knowledge block consists of a set of *entity files*; each entity file consists of a sequence of entities; and each entity has its attributes and properties.

The experimental system which is based on conventional operating systems, has in addition the following design. A *Leonardo individual* is a section of the file system in a computer hosting the individual, that is, one directory and all its sub-directories, with all the files contained in them (with the exception of auxiliary files such as `.bak` files). Each entityfile in the Leonardo sense (i.e., a sequence of entities) is represented by one file in the sense of the file system; this file is a text ('ascii') file adhering to a particular syntax. An *activation* of the individual is obtained by starting a run with a host programming language, where the run is initialized using some of the files contained in the individual. The run usually includes interactions with a human user, but maybe also with robotic equipment, Internet information sources and resources, or other Leonardo individuals. Entityfiles in the individual can be read and written during the activation, for example for storing information that has been acquired during the activation, or for updating the software.

In accordance with the specified goals for Leonardo, as described above, the individual shall be self-contained and be able to model its own structure, and to update it. In that sense the individual is able to *modify itself* during the activation. The individual shall also contain facilities for moving itself, or allowing itself to be moved from one host to another, in ways that are reminiscent of mobile agents ([3]).

The use of directories and files for representing aggregates of Leonardo entities is an intermediate solution. In the longer run we wish to port Leonardo to a persistent software system that is able to represent entities directly, so that the structures and services that are traditionally offered by an operating system and in particular by its file system, can instead be implemented in the Leonardo kernel or platform.

Both the experimental system and the forthcoming persistent system must use a *host programming language*. Functions, procedures, classes, or whatever other building-blocks are used in the host language will be represented by Leonardo entities, and the definition of a function (etc) is expressed in a property of that entity. Our main experimental system has been implemented in CommonLisp; a part of the core has also been implemented in Python. We expect that the persistent system will be based on a language similar to Scheme. Interpretation-oriented languages such as these are the best suited for our approach.

Notation vs. System

The language design and the system design in our approach are strongly interdependent. The language design has come first in the present project, but the system design is by far the largest part of the work and it has arguably the largest novelty value. The main purpose of the present report

[3] `http://en.wikipedia.org/wiki/Mobile-agent`

is to describe the system design, but it is necessary to describe the language design first.

2 Two Examples of LDX

By way of introduction we show two examples of how the Leonardo Data Expression language, LDX, is used in Leonardo. A more detailed specification of LDX can be found in the report "The Leonardo Representation Language ([4]).

The LDX Syntax in a Simple Example

The basic LDX syntax is illustrated in attachment 1 which shows an example of an entityfile called `scandcountries` containing entities representing the Scandinavian countries Denmark, Finland, Iceland, Norway and Sweden. The first entity in the entityfile describes the entityfile itself; the next one represents the type `country`. Each country is specified with an entity representing its capital, named as the English name of the capital city, and with a sequence of strings for the name of the country in its official language or languages. We shall return later to the exact syntax that is used in the textual representation of entityfiles, but already in attachment 1 one can see how an entityfile contains a simple catalogue of its own contents (through the `contents` attribute of the entity `scandcountry`), and how an entity designating a type provides some information about the structure of the instances of that type (through the `attributes` attribute of the entity `country`).

Both the catalogue and the type description is simple in these examples. This illustrates how type information in Leonardo comes in two layers. The first layer, *Catalogue and Type Specification* (CAT) is built into the system kernel and appears in our example. The second layer, *Structure Specification* is considered as a higher design layer and is not included in the kernel.

Notice that entityfiles of this kind are used for expressing both programs and data. Each named unit in a program, such as a function or a procedure, is represented as an LDX entity, with the program code in a property of that entity.

In general, each entity has an attribute called `type` whose value is another entity representing the type of the given entity. Entities will be written in `typewriter font` in this report.

One of the available types is `entityfile`. By convention, the first entity in an entityfile shall always be an entity that has the type `entityfile` (or a subtype of it) and that designates the file. It is used for harboring various structural information about the entityfile.

The operation of *loading* an entityfile is performed by activations, and consists of reading the text file for the entityfile, such as the one shown in Attachment 1, and constructing the corresponding data structures in the activation. The operation of *storing* an entityfile is the reverse operation of re-writing its text file by converting data structures to corresponding,

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

textual expressions. Loading and immediately storing an entityfile has a null effect on its text file.

Cooperating Agents

We proceed now to an example of how a distributed computational process is organized in Leonardo, and how the LDX language is used for representing the control information. Consider the following method description in Leonardo:

```
-------------------------------------------------------
-- method6

[: type method]
[: plan {[intend: t1 t2 (remex: lar-004 (query: makebid))]
         [intend: t1 t3 (query: makebid)]
         [intend: t4 t5 (query: propose-compromise)]}]
[: time-constraints {[afterall: {t2 t3} t4]}]
-------------------------------------------------------
```

This is a plan, i.e. a kind of high-level procedure, for a situation where two separate users have to give their respective bids for some purpose, and when both bids have been received, one user has to propose a compromise. This requires performing the action `query:` three times with different arguments. The time when the first two occurrences are to start is called `t1`; the third occurrence starts at a time `t4` which is defined as being when the first two occurrences have ended. The time when the first mentioned occurrence ends is called `t2`, and similarly for `t3`. The method consists of a set of intended actions, and set of time constraints between them.

This plan is supposed to be executed in a particular individual (called `lar-003` in our specific run of the plan) but the first mentioned action is to be remote executed (therefore `remex:`) in another individual called `lar-004`.

The LDX language is used for representing this plan, or script. In this example there is an entity called `method4` with three attributes `type`, `plan`, and `time-constraints`. The value of the `type` attribute determines what other attributes may be present.

This examples uses more of the LDX expressivity than in the first example. It shows how expressions in LDX may be atomic ones (symbols, strings, or numbers), or may be formed recursively using the operators `<...>` for sequences, `{...}` for sets, `[...]` for records, and `(...)` for forming composite entities. In the example, `(query: makebid)` is a composite entity that has a type and attributes, just like the atomic entity `method4`.

Appendix 2 contains details from an activation using the method shown above, and it illustrates how LDX is used for the control information as the plan or script is executed, and for retaining some of that control information afterwards.

3 Information Structure

The Structure of Knowledgeblocks

The total information in a Leonardo system is organized as a set of knowledgeblocks, and each activation of Leonardo is initialized by loading one specific knowledgeblock in that set. Some knowledgeblocks *require* others, however, so that to load a knowledgeblock one first loads those other knowledgeblocks that it requires, recursively, and then one loads the entityfiles that are specified for the given knowledgeblock itself.

Each knowledgeblock consists of a set of entityfiles. One of those entityfiles represents the knowledgeblock as a whole and contains overall information about it; it is called the *index* of the knowledgeblock. The first entity in the index has the type `kb-index` which is a subtype of `entityfile`, and this entity is used to designate the knowledgeblock as a whole. This means that it can have both attributes that pertain to its role as describing its own entityfile, and attributes that pertain to the knowledgeblock as a whole.

One important use of the knowledgeblock index is to specify where the textfiles for other entityfiles in the same knowledgeblock are stored. The kb index specifies the mapping from entities as understood by Leonardo, to actual file paths in the computer or filestore at hand ([5]). This makes it straightforward to move entityfiles and to redirect references to them, which has a number of uses including that it makes it easy for several individuals to share some of their files.

A few of the entityfiles in a knowledgeblock have special properties or play special roles, besides its index. This applies in particular for ontology files. To the largest extent possible, entities in the core Leonardo system and in its applications are organized in terms of an ontology which is subject to modularization like all other aspects of the system. The kernel contains a 'core ontology', and every knowledgeblock contributes additional entities and links to the ontology, thereby extending the core. Each activation of an individual contains a working ontology that has been formed by loading and integrating the ontology files of the knowledgeblocks that have been loaded.

Entities in a knowledgeblock can be of three kinds with respect to mobility: *software specific, individual specific,* or *host specific*. These are defined as follows. If an individual is moved to another host, then it shall encounter host specific entities of the new host instead of those it had on the old host, whereas software specific and individual specific entities are retained. On the other hand, if a knowledgeblock is exported from one individual to another then only the software specific entities are exported and they will be used with the individual specific entities of the receiving individual. Individual specific information includes the history and experience of the individual; host specific information includes e.g. the locations and properties of databases, printout devices, and other resources that the host can offer to a visiting software individual.

In the present Leonardo design, each entityfile is required to have all its

[5] Some Leonardo individuals are placed on detachable memory devices, such as USB sticks, which means that they can have activations on different hosts without their file structure having been 'moved' in a conventional sense.

members of the same kind in this respect, so that the distinction between software specific, individual specific, and host specific applies to entityfiles as well. The knowledgeblock index specifies only the locations of software specific entityfiles that belong to it. There are separate catalogs for all host specific and for all individual specific entityfiles.

Considerations for the Design of LDX

The Leonardo Data Expression Language (LDX) is a textual representation for information structures, and the above examples have given the flavor of this notation. The details of the syntax are described in a separate memo that is available on the Leonardo website ([6]). The present subsection shall discuss the design considerations that guided the definition of LDX.

The idea of allowing data structures to be expressed as text, and to define input and output of data structures accordingly, was pioneered by John McCarthy with Lisp 1.5 ([7]). It has been adopted in several interpretive or 'scripting' programming languages that are used extensively, such as Perl ([8]) and Python ([9]). It is also characteristic of high level message formats, such as the KQML ([10]). With partly different goals, this tradition has also been continued in the XML family of information representation languages, including e.g. RDF and OWL besides XML itself.

There are several possible motivations for representing information structures textually, in text files or otherwise, and in particular:

1. For persistent storage of the information, between runs of computer programs.
2. For presentation of the information to the user, and for allowing her or him to edit the information.
3. As a message format, for transmitting chunks of information from one executing process to another one.
4. For representation of internal system information, such as parameter settings for particular programs or services.

These alternatives apply regardless of whether the text files are used for representing pieces of code, application data, or declarations, ontologies, or other metadata.

The choice of representation may depend on which of these are the intended uses. In particular, if the second purpose is intended then it becomes important to have a representation that is convenient to read for the human. The poor lisibility of XML was apparently accepted because it was thought that XML coded information should mostly be seen and edited through graphical interfaces, and not directly by users or developers.

In our case, we wish to use LDX for all four of the above mentioned purposes. We also have some other design requirements:

[6] `http://www.ida.liu.se/ext/leonardo/`
[7] `http://www.lisp.org/alu/home`
[8] `http://www.perl.org/`
[9] `http://www.python.org/`
[10] `http://www.cs.umbc.edu/kqml/`

- The notation should be suitable for use in textbooks, research articles, and manuals. This strongly suggests that it should stay as close to conventional set theory notation as possible.
- Since the notation is going to be the basis for an entire computation system, including its programming-language aspects, it must be expressive enough for the needs of a programming language.
- The notation is used for the ontology structure that is a backbone for the Leonardo system. It must therefore be expressive enough for what is needed in ontologies.

These requirements led to the decision of not using an existing language or notation, but to design our own. The following aspects of the LDX language should be emphasized in particular.

1. *The use of multiple bracket types.* Most programming languages use several kinds of parentheses and brackets, such as `(...)`, `[...]`, `{...}`, and possibly others. On the other hand, representations for information structures often use a single kind of brackets, such as `(...)` in Lisp and <...> in XML and other languages in the SGML tradition. This is sufficient in principle, but it makes it necessary to rewrite sets, sequences, and other "naturally parenthesized" structures along the lines of

```
(set a b c)
(sequence a b c)
```

and so on. LRX uses the multiple brackets approach and allows expressions such as

```
{a b c}
<a b c>
```

for sets and sequences, and in addition a few other kinds of brackets. This difference is trivial from an abstract point of view, but it makes surprisingly much difference for the ease of reading complex expressions. Compare, for example, the LDX reprsentation of the plan entity in example 2, which was as follows:

```
------------------------------------------------------------
[: type method]
[: plan {[intend: t1 t2 (remex: lar-004 (query: makebid))]
         [intend: t1 t3 (query: makebid)]
         [intend: t4 t5 (query: propose-compromise)]}]
[: time-constraints {[afterall: {t2 t3} t4]}]
------------------------------------------------------------
```

with a representation of *just the second line* of that information in an XML-style ([11]) single-bracket notation:

```
------------------------------------------------------------
<plan>
   <planstep-set>
      <planstep>
         <intendstep>
            <fromtime>t1</fromtime>
            <totime>t2</totime>
```

[11] `http://www.w3.org/XML/`

```
            <remote-execute>
                <execute-at>
                   <indiv-name>lar-004</indiv-name>
                </execute-at>
                <execute-what>
                   <query-action>
                      <phrase>makebid</phrase>
                   </query-action>
                </execute-what>
            </remote-execute>
         <intendstep>
      <planstep>
-------------------------------------------------------------
```

Even the Lisp-style single-bracket representation is much less convenient to read than the multi-bracket representation:

```
-------------------------------------------------------------
(maplet type method)
(maplet plan
   (set (record intend: t1 t2
           (term remex: lar-004 (term query: makebid)))
        (record intend: t1 t3 (term query: makebid))
        (record intend: t4 t5 (term query: propose-compromise))
           ))
(maplet time-constraints
   (set (constraint afterall (set t2 t3) t4)) )
-------------------------------------------------------------
```

In a historical perspective it is interesting to compare the great interest in legibility issues when programming languages are designed, with the virtually complete disregard for the same issue in the design of so-called markup languages for representing structured information in general.

2. *The use of composite expressions for entities.* LDX is similar to e.g. OWL in that it is based on the use of entities to which attributes and properties are assigned. In the simplest cases, entities are written as identifiers and attributes are expressions that are formed using set, sequence, and record forming operators. However, entities can also be composite expressions that are formed using a *symbolic function* and one or more arguments which are again atomic or composite expressions, for example as in

```
(payment: (membership: (member-number: 1452)
                       (year: 2006) ))
```

for "the payment for the membership during year 2006, by member number 1452", or

```
(b: 356 (remex: lar-004 (query: makebid)))
```

for "the instance of the action `(query: makebid)` that was initiated at time 356 for execution in the individual `lar-004`". Entities formed by composite expressions share the same characteristics as atomic entities, for example that they have a type and can be assigned attributes and properties, and that they can be included in entityfiles with their assignments.

The YAML (Yet Another Markup Language) ([12]) allows assigning attributes to composite structures, but does not make it possible to include such a composite structure as a term in a larger expression.

The use of composite entities has turned out to be very useful in the design of ontologies and other knowledge representations. It is included in the kernel of the Leonardo system and is used in various ways even for representing "system" information in the kernel, as the second introductory example has showed. Another example is for intermediate structures in the version management subsystem. On higher levels of the design, there is an extension of LRX for representing formulas in first-order predicate calculus, in which case composite entities are identified with terms in the sense of logic.

3. *The use of event-state records.* Records are formed in LRX using the notation of the following examples:

```
[date: 2006 10 24]
[quantity: meter 42195]
[quantity: (per: meter second) 46]
```

for the date of October 24, 2006, for the quantity of 42195 meters, and for the quantity of 46 meters per second, respectively. Records differ from composite entities in that they are passive data objects that can not be assigned attributes or properties.

One kind of records, called event-state records, actually allow some of their components to change, but in restricted ways. They are used for representing the current state of an action instance that the system is performing during a period of time, or the current observation state of an event that the system observes. An event record such as (simple example)

```
[move-robot: r14 pos12 pos14 :start 16:22
             :velocity [quantity: mps 4]
             :current-pos [xy-coordinate: 47 12]]
```

where `mps` is an abbreviation for `(per: meter second)`, may represent the current state of an event where the robot `r14` moves from position `pos12` to position `pos14`. The record contains the three *direct arguments* of the operator `move-robot:`, and after them the three *state variables* of the event with the respective labels `:start`, `:velocity`, and `:current-pos`. The values of the state variables, except `:start`, can be changed while the event executes in order to reflect the current state of the robot concerned. When the event ends then the ending time is added as an additional state variable, other state variables are added or removed so that the record becomes a representation of the event as a whole and its final effects, the record freezes, and no further changes are possible in it.

An event-state record such as this may be the value of an attribute of an action-instance entity as formed by, for example, the symbolic function `b:` that was introduced earlier.

The Leonardo Ontology

The Leonardo Ontology is similar to other "top level ontologies" that have emerged in recent years, but it differs from them in two ways. First of all,

[12] `http://www.yaml.org/`

the ontology is intended to be used as the framework for the software system Leonardo, and in particular for its autodescription. It is therefore modular with much smaller granularity than in other ontologies, and it conforms to the knowledgeblock structure of the Leonardo system: each knowledgeblock makes its own contributions to an initial core ontology. Furthermore, the core ontology consists of those constructs that are needed for internal system purposes, although they have been designed in such a way that they also constitute a platform for the ontologies needed by applications. This choice has the effect that the core ontology is small and pragmatic, and that it does not attempt to formulate a general, philosophical basis for its design.

Secondly, the ontology uses the Leonardo Data Expression Language, LDX, which means in particular that ontological terms (i.e., nodes in the ontology) can be composite expressions in the ways that were described earlier. They are not restricted to symbols with more or less mnemonic or structured names. This provides a powerful means of expression in particular for applications.

Beyond these differences, the Leonardo Ontology shares many characteristics with other ontologies, in particular, the requirement that it shall contain the means for describing its own structure.

Additional details about the Leonardo Ontology can be found in Appendix 3.

4 The Leonardo Kernel and Platform

The Structure and Constituents

The Leonardo Kernel consists of four knowledgeblocks, beginning with the *core* which is called `core-kb`. By convention, the names of knowledgeblocks end with "`-kb`". The core satisfies all the requirements on the kernel except version management. In addition there is `chronos-kb` that implements a representation of calendar-level time and of events in the lifecycle of an individual, `config-kb` that is used for creating new copies ("individuals") of the system and for configuring old and new individuals, and finally `syshist-kb` that implements version management. Both reproduction and version management add entries to the system's history of its own activities which is maintained by `chronos-kb`. For example, a so-called synchronization in the sense of version management is treated as an event in the representation provided by `chronos-kb`.

The more basic aspects of self-modification in the system are implemented in `core-kb`, however. This includes, for example, facilities for allowing the user to edit attributes and properties of an entity, and to add entities and entityfiles.

The core part of the Leonardo ontology, called `coreonto`, is an entityfile within the the initial knowledgeblock `core-kb`. Every other knowledgeblock, including the other three kernel blocks, can have their own ontology files that extend preceding knowledgeblocks.

The Core Knowledgeblock, `core-kb`

The following are the contents of the core block, as organized in a number of entityfiles:

- The initial loading or 'bootstrap' machinery. It consists of a few entityfiles that are the very first ones to be loaded when an activation is started, and it prepares the ground for subsequent loading.
- The index file of the core knowledgeblock. (`core-kb`).
- The ontology file of the core knowledgeblock, which is at the same time the core or "top-level" ontology of Leonardo as a whole. (`coreonto`).
- Miscellaneous additions to the core ontology that are needed for historical or other reasons. (`toponto`).
- Definitions of procedures for loading entityfiles and parsing the textual representation of entities. (`leo-preload, leoparse`).
- Definitions of elementary operations on the structured objects in the Leonardo data representation, such as sequences, sets, and records. (`leoper`).
- Miscellaneous auxiliary functions that are needed in the other entityfiles but which have a general-purpose character. (`misc`).
- Major timepoints in the history of the present instance of the system (`mp-catal`).
- Functions for administrating entities, entityfiles, and knowledgeblocks, for example, for creating them and for editing their attributes. (`leo-admin`).
- Functions for writing the textual representation of entityfiles, and for producing the textual representation of Leonardo datastructures from their internal ones. (`leoprint`).
- Definitions for a simple executive for command-line operation of the system. (`lite-exec`).

The entityfile `mp-catal` mostly serves `chronos-kb`, but it is initialized in the core block which is why it is present in this list.

Many of these blocks are straightforward and do not require further comment here; their details are described in the systems documentation. I have already described and discussed the data format for the textual representation of entityfiles. The files for loading and storing that representation (`leo-preload, leoparse, leoprint`) are direct implementations of the data format. Furthermore I shall discuss the ontology, the bootstrap machinery, the machinery for cataloguing entityfiles using knowledgebase index files, and the facility for defining multiple configurations within an individual. Final sections will describe the other parts of the kernel, namely, the facility for administrating and 'remembering' information about calendar-time-level events in the history of a Leonardo individual, and the facility for version management of entityfiles.

The Leonardo Startup Machinery

One of the basic requirements on the Leonardo Kernel is that it shall be able to administrate itself, and as well it shall provide facilities for self-administration of other knowledgeblocks that are built on top of the four knowledgeblocks in the kernel. This self-administration requirement includes several aspects:

- All program code in an implementation shall be represented as entityfiles, without exceptions. This guarantees that general facilities for administration and analysis of Leonardo software can apply even to the initial parts of the bootstrap process.
- Since interactive sessions with the Leonardo system typically involve loading information from the textual representation of entityfiles, modifying their contents, and re-storing those entityfiles, it shall be possible to edit all entityfiles for software in that way as well.
- However, it shall also be possible to text-edit the file representation of an entityfile and load it into an activation of Leonardo, in order for the edits to take effect there.
- In addition, there shall be a version management system that applies to software entityfiles like for all other entityfiles.

The first three of these aspects is implemented using the `core-kb` knowledgeblock; the fourth one using the separate `syshist-kb` knowledgeblock. Notice, however, that the first aspect is a step towards (i.e., facilitates greatly) the fourth one.

The startup process for Leonardo activations is actually a good illustration of how a somewhat complex process can be organized around its symbolic data structures. Appendix 4 describes this in some detail.

Configuration Management

One Leonardo individual may contain the software for a number of applications, for example for simulation, for robotics, for document management, and so on. However it may not be necessary, or even desirable to have all of that software and its associated application data present in a particular activation of the system. The individual should therefore have several *configurations* that specify alternative ways of starting an activation. The startup files that were described above serve to define such configurations. In particular, the `kb-included` attribute specifies which knowledgeblocks are to be loaded when the system starts. Knowledgeblock dependencies whereby one knowledgeblock may require some other knowledgeblocks to be loaded first are supported, and are represented by particular attributes on the knowledgeblocks themselves.

Each configuration may also make some other specifications, for example for extra information that is to be loaded in order to start it. Furthermore, each configuration shall specify its user interface, in the sense of a command-line interpreter, a GUI, and/or a web-accessible service. This is done with the `execdef` attribute on the startup-file that was described in Appendix 4.

The Knowledgebase Index Files

Each Leonardo individual is represented as a directory structure, consisting of a top-level directory and its various subdirectories on several levels, with their contents. In a predecessor to Leonardo, the Software Individuals Architecture, we used fixed conventions for where the entityfiles would be located within the directory structure, and relative addressing for accessing them. This turned out to be too inflexible, and for Leonardo we have a convention where each entity representing an entityfile is associated with the path to where the textual entityfile is to be found.

At first it would seem that this should be one of the attributes of the entity that names and describes the entityfile, and that is the first element in the entityfile. However, it would be pointless to put that attribute within the file itself, since the system needs it in order to find the file so it can load it. One can think of two ways out of this dilemma: either to divide the attributes of an entity into several groups that can be located in different physical files, or to construct a composite entity with the entityfile entity as its argument.

Both approaches have their pros and cons. Leonardo does provide a mechanism for *overlays* whereby one can introduce entities and assign some attributes to them in one entityfile, and then add some more attributes in an overlay, which is a separate file. However, that facility is not part of the kernel, and we are reticent of putting too much into the kernel. Also, overlays require the entity as such to have been introduced first, before the overlay is added. The attribute for the location of an entityfile is needed before the entity itself is available.

We have therefore chosen the other alternative. The following is a typical entity in an index file for a knowledgeblock, such as `core-kb`:

```
------------------------------------------------------------
-- (location: leoadmin)

[: type location]
[: filepath "../../../leo-1/Coreblock/leoadmin"]

@Comment
Loading entityfiles and knowledgeblocks, creating new ones,
etc.

------------------------------------------------------------
```

It defines the location of the entityfile `leoadmin` by introducing a composite entity `(location: leoadmin)` whose type is `location`, and assigning a `filepath` attribute to it([13]). Among the files that occur at the beginning of the startup phase, `self-kb`, `kb-catal` and `core-kb` consist mostly or entirely of such entities.

[13] Actually this attribute is called `filename` in the current system, for historical reasons. This is due to be changed.

5 Other Kernel Knowledgeblocks

Until this point we have described the design of the core knowledgeblock, `core-kb`. The Leonardo kernel also contains three other knowledgeblocks, beginning with `chronos-kb` that enables the Leonardo activation to register events and to have an awareness of the passing of time and a notion of its own history. Based on it there is the reproduction facility, `config-kb`, and the versions management facility, `syshist-kb`.

Both reproduction and version management are essential for the evolution of the Leonardo software through concurrent strands of incremental change in several instances of the system, i.e. several Leonardo individuals. This is the decisive factor for considering these to be an integral part of the system kernel. In addition, by doing so we also provide a set of tools that can be used in applications of several kinds. – The importance of having software tools for version administration do not need to be explained; it has been proven through the very widespread use of tools such as CVS ([14]).

The following are brief summaries of the services that are provided by these knowledgeblocks in the kernel:

Awareness of Time in the Leonardo Individual

The basic contributions in `chronos-kb` are the following:

- A facility for defining and registering significant *timepoints*. Such a timepoint is registered with its date, hour, minutes, and seconds, and it can be associated with the starting or ending of events.
- A facility for introducing *events* in a descriptive sense: the system is told that a particular event starts or ends, and registers that information.
- A facility for defining *sessions* which are composite events corresponding to the duration of one activation of the Leonardo system, and for defining individual events within the session.

All of this information is built up within the Leonardo system, and is maintained persistently by placing it in entityfiles.

System History and Version Management

The system history is a kind of skeleton on which several kinds of contributions can be attached. The first of these is the version management facility which consists of two parts, one that is local within an individual, and one that requires the use of two individuals.

Local version management works as follows. The individual maintains a sequence of *archive-points* which are effectively a subset of the timepoints that are registered by `chronos-kb`. Archive-points have names of the form `ap-1234`, allowing up to 9999 archivepoints in one individual. Each archivepoint is associated with the archiving of a selection of files from one particular knowledgeblock. The archiving *action* takes a knowledgeblock as

[14] `http://www.nongnu.org/cvs/`

argument, obtains a new archivepoint, and for each entityfile in the knowledgeblock it compares the current contents of the file with those of the latest archived version of the same file. It then allocates a new directory, named after the new archive-point, and places copies there of all entityfiles where a nontrivial difference has been identified. The archive-point is an entity that is provided with attributes specifying its timepoint, its knowledgeblock, the set of names for all entityfiles in the knowledgeblock at the present time, and the set of names for those entityfiles that have been archived.

However, the comparison between current and archived version of the entityfile also has a side-effect on the current file, namely, that each entity in the file is provided with an attribute specifying the most recent archive-point where a change has been observed in that particular entity. This makes it possible to make version management on the level of entities, and not merely on entire files, which is important for resolving concurrent updates of the same entityfile in different individuals.

Local version management is useful for backup if mistaken edits have destroyed existing code, but it does not help if several users make concurrent changes in a set of entityfiles. This is what *two-party version management* is for. In this case, there is one 'server' individual that keeps track of updates by several users, and one 'client' that does its own updates and sometimes 'synchronizes'([15]) with the server. Such synchronization must always be preceded by a local archiving action in the client. Then, downward synchronization allows the client to update its entityfiles with those changes that have been incorporated into the server at a time that succeeds the latest synchronized update in the client. If the current entityfile version in the client is not a direct or indirect predecessor of the version that is presently in the server, then no change is made. After that, an upward synchronization identifies those entityfiles whose contents still differ between the server and the client. If the version in the server precedes, directly or indirectly, the current version in the client, then the current version in the client is imposed on the server.

In the remaining cases, the system attempts to resolve concurrent changes in a particular entityfile by going to the level of the individual entities. If that is not sufficient, the user is asked to resolve the inconsistency.

A particular technical problem arises because these synchronization actions require the Leonardo activation to read and compare several versions of the same entityfile. The problem is that normally, reading such a file makes assignments to attributes and properties of the entities in the file, but for synchronization purposes one does not wish the definitions in one file to replace the definitions that were obtained from another file. This problem is solved using composite entities, as follows: The procedure for reading an entityfile in LDX format has an optional parameter whose value, if it is present, should be a symbolic function of one argument. If it is absent then the file is read as usual. If it is present, on the other hand, then that function is applied to each entity that is read from the file, obtaining a 'wrapped' entity, and the attributes and properties in the file are assigned to the wrapped entity. After this, the comparisons and updates can proceed in the obvious way.

We have now seen two examples of how symbolic functions and composite

[15] This is the usual term, although it is of course a terrible misuse of the word 'synchronize'.

entities have been useful even for internal purposes within the kernel. This illustrates the potential value of reorganizing the overall software architecture so that certain, generally useful facilities are brought into, or closer to the system kernel, instead of treating them as specific to applications.

Configuration and Reproduction of Individuals

One of the important ideas in Leonardo is that the system shall be self-aware, so that it is able to represent its own internal state, to analyze it and to modify it, and it shall be able to represent and "understand" its own history. Furthermore, all of this shall occur in persistent ways and over calendar time, and not only within one activation or "run" of the system.

We believe that these properties are important for a number of applications, but in particular for those that belong to, or border on artificial intelligence, for example for "intelligent agents". A system that acquires information during its interactions with users and with the physical world, and that is able to learn from experience for example using case-based techniques, will certainly need to have persistence. It does not make sense for the system to start learning again each time a new activation is started. It is then a natural step to also provide the system with a sense of its own history.

One must then define what is "the system" that has that persistence and sense of its own history. What if the software is stored in a server and is used on a number of thin clients that only contain the activations? What if several copies of it are taken and placed on different hosts? What if a copy of the system is placed on a USB stick so that it can be used on several different hosts?

In the case of Leonardo, the answer is in principle that each individual is a self-contained structure that contains all of the software that it needs. Different individuals may contain equal copies of that software, but in addition each of them contains its own history and its own "experience". However, it is also perfectly possible for each individual to modify its software so that it comes to differ from the software of its peers.

What if additional copies (individuals) are needed, for example because additional persons wish to use the system? The simplest solution is to have an archive individual from which one takes copies for distribution, but in any case that archive individual will change over time, so a notion of version or generation of the entire individual will be needed. But more importantly, separate strands of the Leonardo species may develop in different directions, and a particular new user may be more interested in obtaining a copy of his friend's Leonardo rather than one from the archive.

In principle, a new individual that is obtained from a Leonardo individual by copying its software but erasing its history and other local information, is to be considered as an "offspring" and not as a "copy". If the copy is perfect and all history is preserved in it, then it shall be called a "clone". The administration of clones offers additional problems that will not be addressed here.

For offspring, the following conventions are adopted. The making of an offspring from an individual is to be considered as an action of that individual, and is to be recorded in its history. Each individual has a *name*, and the offspring of a particular individual are numbered from 1 and up. No individual

is allowed to have more than 999 offspring. The first individual under this scheme was called `lar`, and its direct offspring are called `lar-001`, `lar-002`, etc. The offspring of `lar-002` are called `lar-002-001`, `lar-002-002`, and so forth. The abbreviation `lar` stands for "Leonardo Ancestry Root".

The overall convention for the population of Leonardo individuals is now that new individuals can only be produced as offspring of existing ones, so that the parent is aware of the offspring being produced and so that no name clashes can occur in the population. Additional information about when and where offspring are produced is of course valuable, but can be considered as add-on information.

Notice in particular that version management information is not inherited by offspring, and they start with an empty backup directory as well as an empty memory of past events.

In principle, each new individual should obtain a copy of all the software of its parent. In practice this is quite inconvenient when several individuals are stored on the same host; one would like them to be able to share some of the software files. This has been implemented as follows: Each individual may identify another individual that is known as its "provider", and when its index files specify the locations of entityfiles, they may refer both to files in its own structure, and files in its provider. An individual is only allowed to update entityfiles of its own, and is not supposed to update entityfiles in its provider ([16]). When a new individual is created, then it is first produced with a minimal number of files of its own, and it relies on its parent as its provider for most of the entityfiles. After that, it is up to the offspring to copy whatever software it needs from its provider to itself, until it can cut that umbillical cord. Only then is it in a position to migrate to other hosts. Besides, given adequate software, it may be able to import knowledgeblocks and entityfiles from other individuals and not only from its parent.

What has been said so far applies to Leonardo-specific software. In addition, applications in Leonardo will often need to access other software that is available in the individual's host for its current activation, for example text editors and formatters. The kernel contains a systematic framework for administrating this.

Facilities for reproduction of individuals were first developed in the earlier project towards the *Software Individuals Architecture*. In that project we considered reproduction and knowledge transfer between individuals to be very central in the architecture, besides the abilities for self-modelling. In our present approch reproduction has been relegated to a somewhat less central position, due to the experience of the previous project.

Other Facilities in the Kernel

The four knowledgeblocks in the kernel also contain a number of other facilities that have not been described here. In particular, there is a concept of a "process" in a particular sense of that word. Leonardo processes are persistent things, so they exist on calendar time and not only within one activation of the system. Each process has its own subdirectories where it

[16]This restriction is not enforced at present, but users violate it at their own risk.

maintains its local state between activations, and each activation is an activation *of* one particular process. Each process can only have one activation at a time, but different processes can have activations concurrently.

6 Platform Facilities

The next layer in the Leonardo software architecture, after the kernel, is called the *platform*. This layer is under construction and is intended to be open-ended, so that new contributions can be added continuously as the need is identified and the implementation is completed. The following are some platform-level knowledgeblocks that exist and are in use at present.

Channels

Leonardo channels are a mechanism for sending messages between individuals, for the purpose of requesting actions or transmitting information. Each channel connects two specific individuals for two-way, ansynchronous communication and is associated with a number of attributes, including one specifying the data format to be used for the messages. The LDX data format is the default choice.

Communicable Executive

The initial example in this article describing the interactions between two Leonardo individuals was executed using our *communicable executive* (CX). The basic command-line executive in the kernel is not sufficient for it. CX performs incessantly a cycle where it does three things:

- Check whether an input line has been received from the user. If so, act on it.
- Check what messages have arrived in the incoming branch of the currently connected channels for this individual. If so, pick up the messages and act on them.
- Visit all the currently executing actions in the present individual, and apply an update procedure that is attached to each of them. This procedure may perform input and output, update the local state of the action, and terminate the action with success or failure, if appropriate.

The communicable executive is a natural basis for several kinds of applications, including for some kinds of robotic systems, dialog systems, and simulation systems.

7 Extending the Kernel into Applications

Until this point we have described the overall design of Leonardo and in particular its present kernel. We have already a number of applications that operate on the basis of this kernel, and that are represented by around

fifteen knowledgeblocks in various stages of completion. Some are in daily use; some are still being developed.

In line with the plans for the Leonardo project as a whole, it is our intention to complete a sufficient number of these and to verify that it has been possible to implement both the kernel and those applications without any major duplication of representation schemes and services. It is still too early to report on the confirmation (or refutation) of that hypothesis, and this must be a topic of a later publication. At any rate we have not seen any signs yet of it being refuted, and the outlook is promising.

8 Discussion and Conclusions

Design Considerations

Some aspects of the rationale for the design of the present Leonardo system have been explained from time to time in the previous sections. Overriding aspects, such as the absence of datatype declarations and of a conventional database system in the kernel will be discussed in the subsection on software consolidation, below. Here we shall mention one design consideration of more specific nature.

Namespace for entity names. It may seem strange that the names of entities are global, and there is no explicit facility for having multiple namespaces. The reason for this is that there is a more general facility that can also provide the need for namespaces, namely the use of symbolic functions and composite entities. Suppose, for example, that one would like to use the symbol `key` both for the key for a lock, and for the key on a keyboard, and that these are introduced in two different knowledgeblocks, `security-kb` and `equipment-kb`. One could then simply represent them as

```
(local: key security-kb)
(local: key equipment-kb)
```

Other choices of qualifiers are also possible; one might represent the former kind as

```
(usedfor: key lock)
```

but notice that the term `lock` may also be ambiguous. If the ambiguous identifier, such as `key` or `lock` appears in a context, then the proper meaning of it there has to be determined by inference.

History and Current Status of the Experimental Implementation

The design for Leonardo started in early 2005. It was based on the earlier experience with the Software Individuals Architecture (SIA), and with several earlier systems before that. The SIA was used as the platform the a major part of the Linköping-WITAS Robotic Dialog Environment, RDE ([17]), which contributed valuable background for the present system.

[17] `http://www.ida.liu.se/ext/casl/`

During the almost two years of Leonardo development we have tried to make 'laboratory notes' documenting what steps were taken, what design changes were made, and so on. We shall study the possibility of extracting a more concise account of essential design decisions and design changes from these laboratory notes.

The present author uses a Leonardo-based software application as his standard tool for the preparation of articles and other documents and for website pages, including the extensive CAISOR website ([18]). This is a way of checking that the system is always kept operational while it is being revised and extended continuously.

The first additional user of Leonardo, besides the present author, started using the system in October, 2006.

The Need for Software System Consolidation

The main goal of the Leonardo project, as we stated initially, is to explore the possibility of obtaining a much simpler design of the overall software system in a computer, in particular by reorganizing and realigning its major parts so as to eliminate duplication of concepts and of software facilities. It is not yet possible to evaluate the concrete, experimental Leonardo system design against that goal, but it is possible to identify how the new design relates to some of the concrete redundances in conventional systems. They are as follows:

Duplication of procedural language between operating system (shell scripts) and programming languages. In Leonardo there is a host language which may vary between generations of the system, but which shall in any case be a language of the 'interpretive' or 'script' type, such as Scheme, Python, etc. The Leonardo kernel provides the command-script situation, and the language can be extended with more facilities, and restricted using e.g. type system, in order to satisfy the needs of other usage situations.

Duplication of notations and systems for type declarations of data structures, between programming languages, database systems, communication systems e.g. CORBA, etc. The two-layered approach to the type system in Leonardo was explained in the beginning of section 2. Exactly because the type system is not built into the system kernel, we foresee that it shall be possible to design it in such a flexible way that it can satisfy the varying needs of several kinds of contemporary type systems. This is of course one aspect of the main design hypothesis that was stated at the beginning of the present report.

Scripting languages in various software tools, for example spreadsheet systems, webpage languages such as Javascript, etc.. The idea is that such tools ought to be implemented based on the Leonardo kernel and inherit its facilities, including in particular the use of the host language.

Duplication between the file-directory system and database systems. Although the present, temporary implementation of Leonardo is based on a conventional OS and makes fairly extensive use of its file system, the long-term idea is to replace it with an implementation of entities and aggregates

[18] `http://www.ida.liu.se/ext/caisor/`

of entities that is done on directly on the base software. This new information structure shall then subsume what the file-directory system does today.

In the continued work on Leonardo we are going to build a number of applications for the purpose of obtaining additional experience with these and other aspects of duplication. At the same time we shall be vigilant about what new duplications may arise as the system and the applications grow in size and complexity.

References

Due to the character of this material, most of the references are to websites that provide information about a particular language or system. These references have been placed in footnotes on the page where the reference occurs.

References to published articles and released reports from the Leonardo project can be found on the project website, ([19]). References to published articles from the preceding Software Individuals Architecture project (SIA) can be found on its past project website ([20]).

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

[20] `http://www.ida.liu.se/ext/caisor/systems/sia/page.html`

Appendix 1: A very simple entityfile

The following is a very simple example of an entityfile, for the purpose of illustrating their charateristics.

Notice in particular how entities are separated by a line of dashes. This line is syntactically significant, and it has been chosen because of how it facilitates reading the file. Conventional ways of designing a syntax for complex objects would rather have resulted in a notation where the separation between two entities consists of an "end" symbol immediately followed by a "begin" symbol for the next entity. We maintain that the dashed-line syntax makes it much easier for the reader to orient herself or himself in the text. Admittedly it does not allow recursive nesting of entity-descriptions, but that is not really needed. (It is possible however to subdivide an entityfile into sections, where each section begins with a line of equality-signs instead of dashes. This is probably as much hierarchy as we would need within one and the same entityfile).

```
------------------------------------------------------------
-- scandcountry

[: type entityfile]
[: contents <scandcountry country denmark finland iceland
   norway sweden>]

------------------------------------------------------------
-- country

[: type thingtype]
[: subsumed-by spatial-entity]
[: attributes {ownnames capital}]
[: create-proc cre-country]

------------------------------------------------------------
-- denmark

[: type country]
[: ownnames <"Danmark">]
[: capital Copenhagen]

------------------------------------------------------------
-- finland

[: type country]
[: ownnames <"Suomi" "Finland">]
[: capital Helsinki]

------------------------------------------------------------
-- iceland

[: type country]
[: ownnames <"Island">]
[: capital Reykjavik]

------------------------------------------------------------
```

```
-- norway

[: type country]
[: ownnames <"Norge" "Noreg">]
[: capital Oslo]

------------------------------------------------------------
-- sweden

[: type country]
[: ownnames <"Sverige">]
[: capital Stockholm]

ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
```

Appendix 2: A Session with two Interacting Agents

Section 2 in this article described the representation of an action plan in LDX. Consider an interactive session where this plan is put to use. There are two open command-line windows on the computer screen or screens, one for each of the two Leonardo individuals `lar-003` and `lar-004` which may be located on the same computer or on two different ones. The interactions on `lar-003` go as follows, after the obvious startup of the system:

```
025-> adg (achieve: demo example A)

026-> selmeth method6
```

The command prompt consists of an interaction number and the characters `->`; user input consists of a command often followed by an argument. The `adg` command requests the system to adopt a particular goal which is characterized by the command's argument. In the full system this should lead to a process for obtaining a plan, either by planning from first principles or by retrieving a plan from an archive. However, in this example we have suppressed the planning phase using the second command, `selmeth`, which simply instructs the most recently introduced goal which plan to use. The plan starts to execute when the user enters the command `seg`, for 'start execute goal':

```
027-> seg
    > (adogoal: 25 (achieve: a b c))

029=>
----> Please make your bid: 444

030=> [28] Succeed, result: 444

031=>
032=>
----> Please propose a compromise: 666
033=> [31] Succeed, result: 666
Completed goal: (achieve: a b c) adopted at: 25
```

The following is what happens. When the user types in `seg`, the action `(query: makebid)` starts to execute in the individual at hand, which has the effect of displaying the prompt `Please make your bid:` in the individual's user dialog. At the same time, the first action in the plan starts to execute remotely, in the other individual, where it displays a similar prompt on its screen. (The seemingly redundant command prompts reflect intermediate stages in the system's internal processing).

The wordings of the prompts is obtained because the arguments of `query:` are separately defined entities that have the wording as an attribute. The following is the definition of the entity `makebid`:

```
------------------------------------------------------------
-- makebid

[: type output-phrase]
[: englishphrase "Please make your bid:"]
[: swedishphrase "Var god ge Ditt bud:"]
```

The user for the first individual answers the prompt with the value `16200`, which counts as interaction `029`, and the system confirms completion of that action in interaction `030`. The first individual also receives the value from completion of the action in the other individual, in interaction `031`. The top level executive listens to input both from the user and in channels from other agents/individuals.

The completion of the first two actions allows `lar-003` to start performing the third action in the plan, leading to interaction `032`, after which the goal is reported as completed in interaction `033`.

The information about what actions were performed, for what reason, and with what results, is represented as LDX data structures and is therefore available for inspection and for further processing. The command `log`, for 'list old goals', displays the current information about the goal used above, as follows:

```
050> log
  (adogoal: 25 (achieve: a b c))
  [: plan-name method6]

  method6
  [: type method]
  [: plan {[intend: t1 t2
                (remex: lar-004 (query: makebid))]
           [intend: t1 t3 (query: makebid)]
           [intend: t4 t5 (query: propose-compromise)]}]
  [: time-constraints {[afterall: {t2 t3} t4]}]
```

This is the actual plan that the system uses for achieving the goal, and since in our case we told the system what plan to use, it is not surprising that this is the same structure as in the method except for the addition of the `:done` elements that represent that the action-intention in question has been completed.

Notice that the logs are also expressed in LDX. The command `loa`, for 'list old actions' displays the actions that were performed in the first individual, as follows:

```
075-> loa
  (b: 27 (remex: internal-ch-02 (query: makebid)))
  [: type action-instance]
  [: state [requested:]]
  [: subactions <>]
  [: outcome [result: 555]]
  [: endtime 32]

  (b: 28 (query: makebid))
  [: type action-instance]
  [: state [result: 444]]
  [: subactions <(b: 29 (ask: makebid))>]
  [: outcome [result: 444]]
  [: endtime 31]

  (b: 29 (ask: makebid))
```

```
[: type action-instance]
[: subaction-of (b: 28 (query: makebid))]
[: outcome [result: 444]]
[: endtime 30]

(b: 31 (query: propose-compromise))
[: type action-instance]
[: state [result: 666]]
[: subactions <(b: 32 (ask: propose-compromise))>]
[: outcome [result: 666]]
[: endtime 34]

(b: 32 (ask: propose-compromise))
[: type action-instance]
[: subaction-of (b: 31 (query: propose-compromise))]
[: outcome [result: 666]]
[: endtime 33]
```

The functions `adogoal:` and `b:` are further examples of functions that form composite entities. The function `b:` takes two arguments, namely a timepoint and an action, and forms an entity for the action instance that is/was invoked at the time given in the first argument. The function `adogoal:` is similar but it forms a goal instance from a timepoint and a goal, representing the particular goal instance that results when the goal is adopted at a particular timepoint.

Actions are hierarchical, so actions can have subactions, or more precisely, each action instance can have sub-action-instances. In our example, a `query:` action invokes an `ask:` subaction that makes the prompt and receives the answer. If the answer is malformed then `query:` asks the user again until a correctly formed answer is obtained.

Each action instance has a starting time and an ending time, and is *executing* between those times. The executive in a particular individual visits all currently executing action instances cyclically and applies an update procedure for each of them; the update procedure is determined by the 'verb' or operator in the expression for the action, for example `query:`. It is therefore straightforward to define actions that map incoming sensor data to outgoing actuator data in each cycle during their execution period. Our example here does not illustrate this possibility.

When an action instance or goal instance terminates, it obtains an `outcome` attribute and an `endtime` attribute. The latter is the timepoint of termination. The `outcome` attribute represents whether the action *succeeded* or *failed*, using records beginning with `result:` and `fail:`, respectively. Result records can report a 'value' that results from the action, as well as ancillary information; fail records can report the character of, and possibly the reasons for the failure. The outcome of an action is reported to the superaction from which the current action was invoked, or the goal instance invoking it, or the other agent invoking it in the case of remote execution, or a combination of these.

Pursuit of a goal is straightforward if there is an appropriate plan and if all the actions in the plan succeed. If some action fails, and unless a remedy for the failure has already been defined in the plan, then replanning or resort to the user must follow. Replanning has not been implemented in the current

system, but it is of course on the agenda for future work.

Appendix 3: The Leonardo Ontology

Figure 1 shows the top-level structure of the Leonardo Ontology. Two attributes of particular importance are shown in the diagram as arrows, namely the `type` attribute (black v-type arrow) and the `ako` attribute, or subsumption, which is shown as a red solid arrow. (Here "ako" stands for "a kind of", as usual). The type attribute is obligatory for all entities, and its value is always another entity. The value of the subsumption attribute shall be a set of entitites, although the entitites in the diagram have at most one member in their subsumption attribute.

The graph formed by the `type` attribute assigns an *order* to each entity; the order being a non-negative integer. The type of a zero-order entity shall again be zero-order; the type of a *k*-order entity shall be *k-1* if *k* is 1 or greater. Zero-order, first-order, and second-order entities are displayed with blue, yellow, and salmon color, respectively.

Four zero-level entities are used as roots for structures of higher-order entities, namely `thingtype`, `qualitytype`, `spacetime-type`, and `descriptortype`. Each of these structures is a flat tree with respect to the *type* attribute, but it can contain more structure using the *ako* attribute.

The `qualitytype` hierarchy is easy to grasp using figure 1 and the more detailed diagram in figure 2. For example, `light-red` is a kind of `red`, and both `light-red` and `red` are colors, so that their `type` attribute is `color`. Furthermore, `color` is a kind of `visible-quality` which in turn is a kind of `quality`. It is clear that the `ako` relation is transitive, and that the `type` relation is not.

All of `color`, `visible-quality`, and `quality` have the same type, namely `qualitytype`. That last entity is synthetic, in the sense that it is needed for the logical coherence and proper operation of the system, and it does not have any particularly strong intuitive motivation.

The tree rooted at `thingtype` is similar in the sense that it contains both first-order and second-order entities. In one part of the tree the second-order entities are tangible ones, such as persons and physical objects, as illustrated in figure 3. It is clear how the first-order entities there have a natural subsumption structure, and how the second-order entities donot, although they may have other structures such as a `partof` structure. Another part of the tree contains actions and events, and there the intuition does not provide the same strong guidance. We have chosen however to use a two-level structure for events that is analogous to the one for physical entities.

The main difference between the `qualitytype` structure and the `thingtype` structure is therefore that the former admits a subsumption structure in its second-order entities, and the latter one does not. Within the `thingtype` structure there are two main subdivisions, for individual-entities and for events ([21]). Individual entities are subclassified into tangibles, intangibles (for example ideas, plans, goals), and social entities. Soccer teams, corporations, and editorial boards are examples of social entities. Dinner parties are not; they are examples of events.

[21] One may debate whether maybe the entity `thingtype` should be split up into two entities, namely, one for individual objects and one for events. Considerations such as this one are not very important, and a change of the design in such respects can be done without much affecting the total structure.

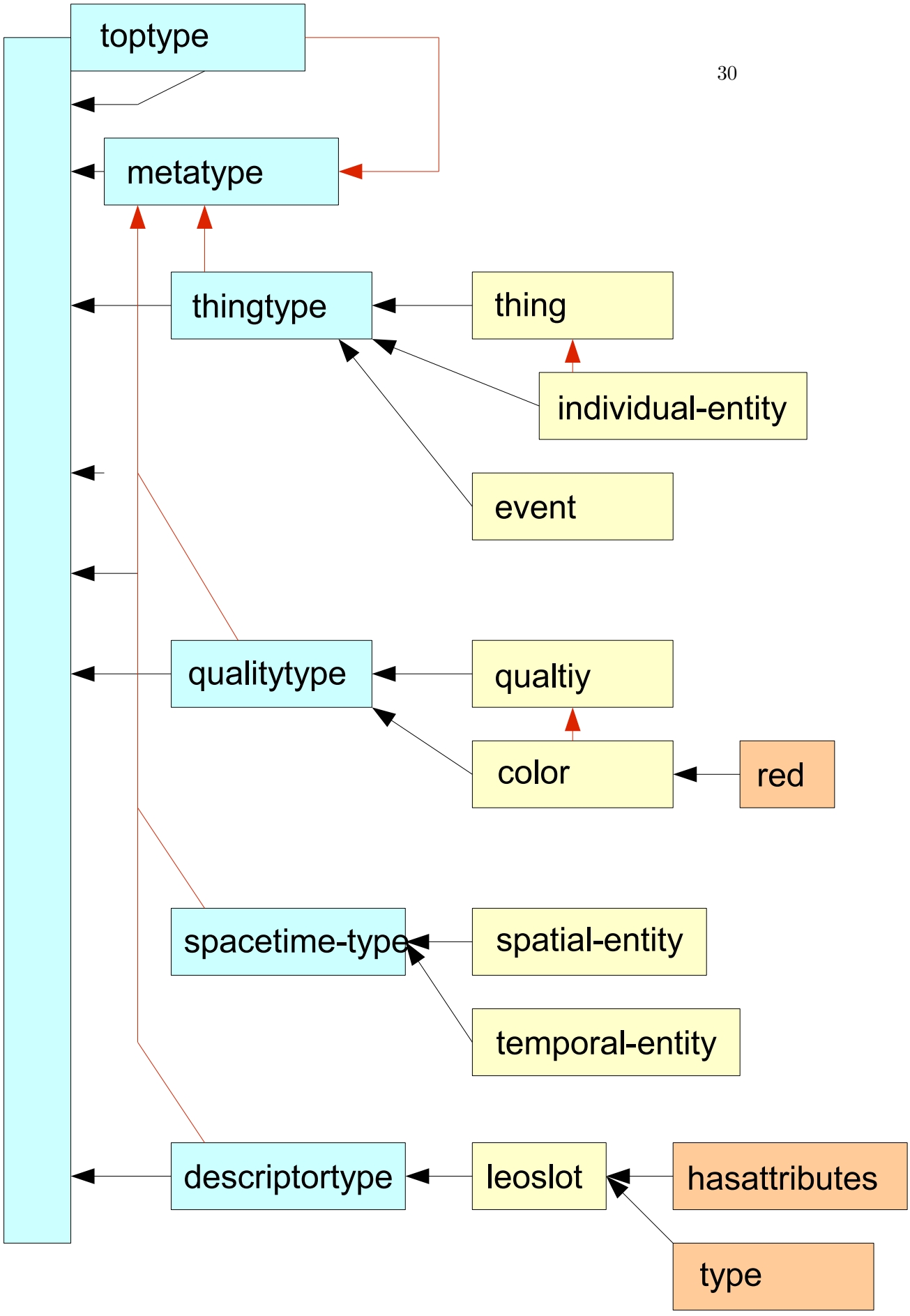

Figure 1: Top level of Leonardo ontology

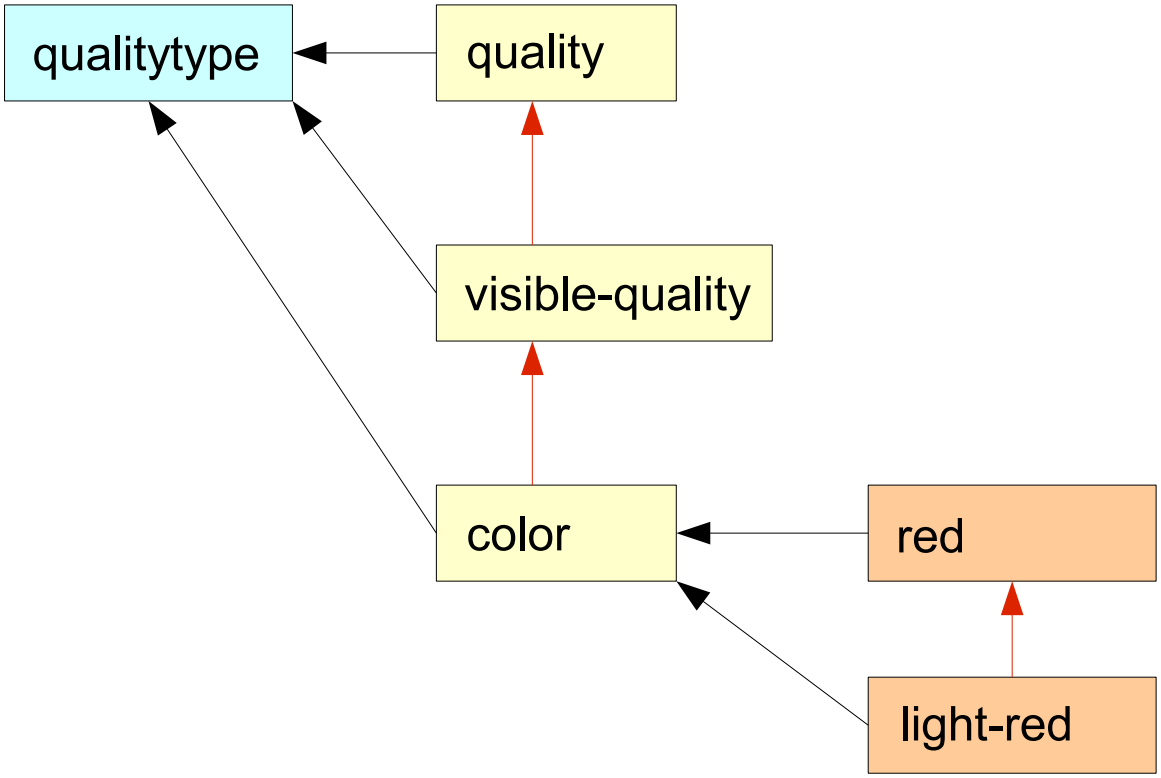

Figure 2: The qualitytype subontology

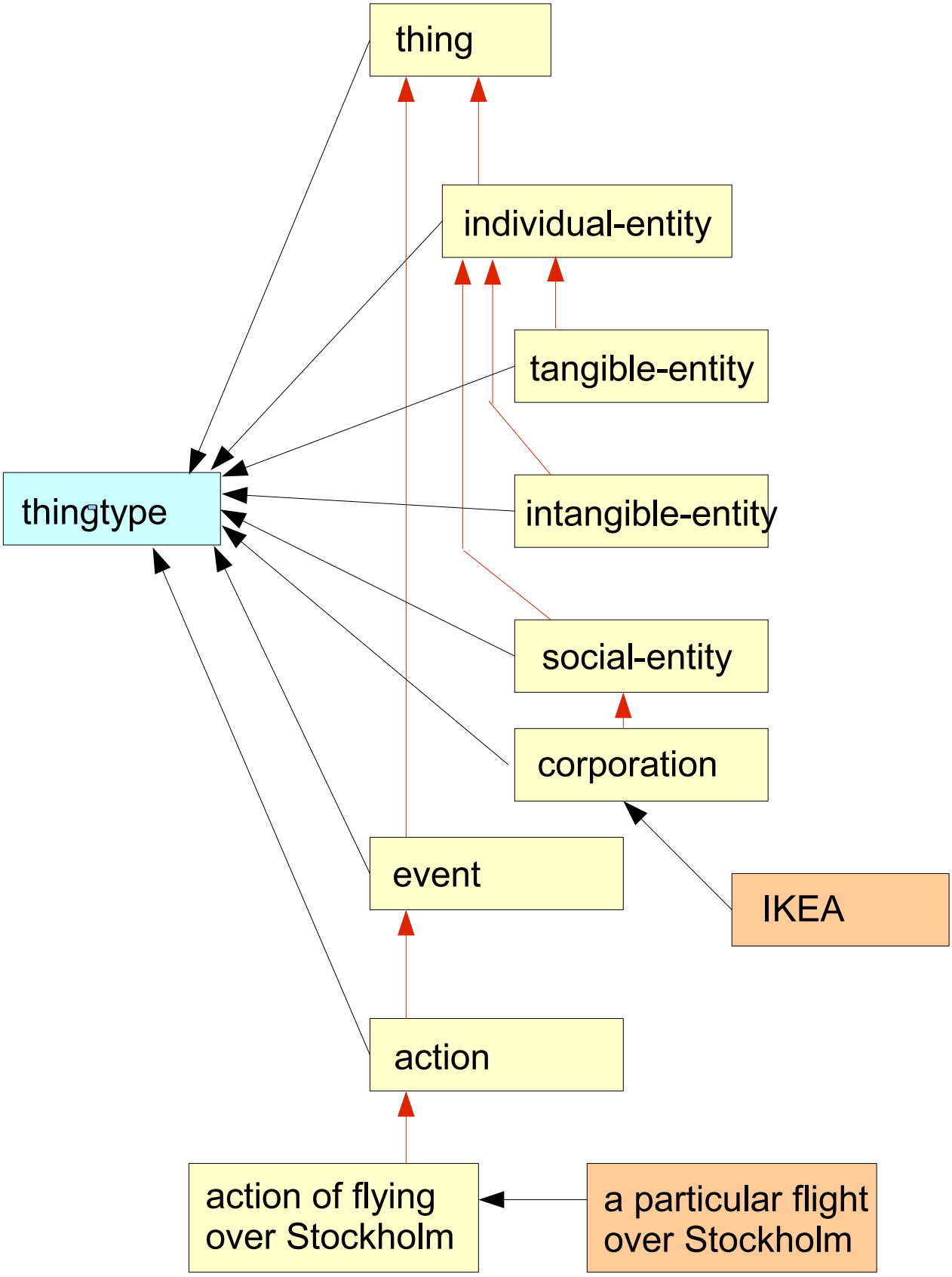

Figure 3: The thingtype subontology

Tangible entities include persons and physical objects. Within intangible-entities we have (presently) the following subtypes: abstract entities (e.g. formulas, triangles), cognition entities (e.g. plans), communication entities (e.g. journal articles, websites), computation entities (e.g. file directories), and leonardo entities (e.g. entityfiles).

Spatial and temporal concepts are organized in the third entity tree which is rooted in `spacetime-type` Finally, there is a fourth entity tree that is rooted in the entity `descriptortype`. It contains for example the type `leoslot` that has the tags for attributes and for properties as some of its instances.

None of the zero-order entities is really needed for applications; they are only there so that the system's self-description is complete and coherent. In particular, the `type` attribute of all zero-order entities is `toptype`, and this entity is zero-order and therefore has itself as its type. In the subsumption dimension, the top-level element among the zero-order entities is called `metatype`, and all other zero-order entities are directly or indirectly subsumed by it.

The entire ontology of the core knowledgeblock, `coreonto`, contains around 35 ontology nodes, all represented as entities. The top level structure that has been described here is included among them([22]). In particular there are a number of entitytypes that are used in the information that the core knowledgeblock organizes itself. This includes types such as `leo-individual`, `os-command`, and `startup-file`.

The ontology represents a tradeoff between two opposing tendencies: in the interest of simplicity we would like to restrict the ontology of `core-kb` to those concepts that that knowledgeblock needs itself, but on the other hand we also want to start with a core ontology that is flexible enough so that the needs of other knowledgeblocks and of applications can be accomodated as extensions in a natural way.

It is not our intention to build yet another ontology, beyond all those that exist already, and the plan is incorporate one or more existing ontologies within the top level that has been described here, and whose design is dictated by the overall systems design. We expect that some adjustments of the top level structure will be needed in this process.

[22] except for obvious examples, such as the 'color' part

Appendix 4: Startup of Activations

Problem and Approach

It is particularly attractive to implement Leonardo in interpretive and 'script'-type programming languaes, such as CommonLisp and Python, but then the implementation of the startup machinery along the lines that were described in section 4 of the article, involves an interesting chicken-and-egg problem. In principle, you want the interpreter of the programming language to be able to read, parse, and digest textual entityfiles, and if an entityfile contains e.g. function definitions then those definitions shall take effect along the way when the file is loaded. More generally, if an entityfile contains an executable expression in the programming language, then that expression is to be executed, since typically the way to define a function is to execute an expression that stores the definition.

Consider now the very first entityfile that is going to be loaded into the interpreter at hand (CommonLisp, Python, etc) in order to start the bootstrap process. The interpreter has been invoked with that file as its argument, and that file is going to add other entityfiles that are needed. If this file is edited and then loaded into the Leonardo system, for example for program analysis or for version management, then the bootstrap process will start again.

Clearly the system needs to be able to make a distinction between 'hot' and 'cold' loading of software entityfiles, where some of the commands in the file are only executed during 'hot' loading. In our (CommonLisp) implementation we obtain this property as a side-effect of a more general facility.

The general facility is as follows. Entityfiles are seen by the user as text files in LDX format which has already been introduced. However, when an entityfile is *stored* then actually two versions of the file are written: the standard one that the user sees, and a "compiled" one consisting of plain Lisp code (i.e., S-expressions), which means that the Lisp interpreter can read the compiled file even before any definitions have been made to it. The primary purpose of this is to produce a faster-loading version of the entityfile.

Normally, loading the "source" (LDX) version and the "compiled" version of an entityfile has exactly the same effects on the executing activation of Leonardo. However, specific entity properties that contain executable program code (CommonLisp code, in our case) can be marked with a special flag having the effect that the code is only executed when the compiled file is loaded, and not when the source file is loaded. Moreover, the index file for a knowledgeblock, which contains information about where the files in that block are located, also contains information about which files shall be loaded from their compiled version or their source version *when they are loaded as part of their knowledgeblocks*. A separate command by the user to load a particular entityfile will always load the source version.

The first entityfiles that are loaded during the bootstrap of a Leonardo activation are therefore marked for compiled-version loading, and they contain a few expressions that only execute in compiled-version loads. This is sufficient for getting the bootstrap process to work as intended. When the user wishes to modify some of the code in those early-session files, she will

typically text-edit the source version of the entityfile, load it into Leonardo, and immediately store it.

Details of the Startup Process

The startup or 'bootstrap' process is actually a nice illustrative example of the declarative style of system design in Leonardo, and we shall therefore describe it in some detail. The reader that is not so interested in the technical details is advised to skip to the beginning of section 5, a few pages ahead.

Suppose `minileo` is the entityfile that is first read by the interpreter of the host programming language, and that it will only load the minimal knowledgeblock, `core-kb`, and no other knowledgeblocks. The following are the first entityfiles that are read during startup:

```
minileo
bootfuns
selfdescr
self-kb
kb-catal
core-kb
coreonto
  ...
```

In this sequence, `core-kb` is the first file for loading the core knowledgeblock and from there on, the system reads the entityfiles that are defined by the knowledgeblock. They were listed in section 4, page 21, and are loaded in exactly the same way as for all other knowledgeblocks. The five simple files preceding `core-kb` are the only ones that have a special role. Going backwards, `kb-catal` is a catalog containing the physical locations (directory and filename) for the index files of all the knowledgeblocks in the individual at hand. This is needed so that the system can look up `core-kb` in order to load it, as well as for loading any other knowledgeblock. Before it, `self-kb` is a catalog of those entityfiles whose contents are specific to the present individual and/or host, in particular, the location of `kb-catal`. Again before, `selfdescr` is a small entityfile containing the name and ancestry of the individual at hand, so that it 'knows' what it is called and from where it was generated. The locations of `selfdescr` and `self-kb` are hardwired into the software.

The file `bootfuns` contains function definitions for functions that are used in the immediately following files, and it ends with an executable expression that is only executed when the compiled version of `bootfuns` is loaded. It is this expression that loads `selfdescr`, `self-kb`, and `kb-catal` as individual files.

The file `minileo`, finally, is a small file containing a compiled-mode executable expression that does two things. First it loads `bootfuns` in compiled mode, which defines a number of functions and then loads `selfdescr`, `self-kb`, and `kb-catal`. Then it invokes a particular function `leoboot` of no arguments; this function is defined in `bootfuns` and has the effect of loading the knowledgeblock required by the startup file, as well as performing some simple administrative duties.

Data-driven initial startup

This structure of interacting loads and invocations has been designed so that it shall be easy to reconfigure the system, for example when moving a Leonardo individual to another host, or when creating an offspring of an individual. (An offspring inherits the software that is contained in an individual, but not its history of past events). Notice the simplicity of the definition of the entity `minileo` that was used in the example:

```
[: type startup-file]
[: contents <minileo>]
[: batname "minileo"]
[: kb-included <core-kb>]
[: bootfile "../../Coreblock/cl/bootfuns.leos"]
[: execdef lite-exec]
```

Its type, `startup-file`, is a specialization of the type `entityfile`. All aspects of startup are datadriven using its attributes, so other configurations can be defined by setting up other entityfiles of the type `startup-file` and providing them with other values for the attributes. In particular, the `bootfile` attribute specifies where to find the first file to be loaded after itself, which in this case is `bootfuns`. The `kb-included` attribute specifies a list of knowledgeblocks that are to be loaded, which in this case is only `core-kb`. Finally, the `execdef` attribute specifies which entityfile to load in order to obtain the configuration's user interface. The `bootfile` attribute specifies the exact position of the boot functions relative to the current one.

The interpretation of the attributes in a `startup-file` entity is not performed by a central procedure, but by procedures that are attached to the entity itself, in the following way. The following executable expression is added in the compiled version of such a file.

```
(setq *myconfig* 'minileo)
(load (get *myconfig* 'bootfile))
(leoboot)
```

or, in translation to conventional programming-language syntax:

```
myconfig := 'minileo;
load(myconfig:bootfile);
leoboot()
```

This expression is identically the same in any compiled startup-file, except that of course it is the name of the file at hand that appears in the assignment on the first line. The expression is generated by a procedure attached to a handle that is provided for every subtype of `entityfile` and that allows the addition of extra material at the end of a 'compiled' entityfile.

When the Leonardo system starts with this configuration, it first loads the compiled version of `minileo`, which contains `CommonLisp` assignment statements for the entity attributes, followed by the statements just mentioned. The assignment statements assign in particular the value of `(get *myconfig* 'bootfile)` which determines which file is going to be loaded next. There may be a choice of several such files, but any file that is used in this way must define the function `leoboot`. This arrangement makes it possible to define entirely different startup sequences with just a few attribute assignments.

The special procedure for writing startup-files makes one additional thing (in the Windows environment): it generates a `.bat` file that invokes the CommonLisp interpreter with the compiled startup-file at hand as the initial load. The `batname` attribute specifies the name of that `.bat` file.

In this way the construction of configurations is automatic and datadriven, and is done within the general framework of entityfiles and the LDX representation language. This is very convenient when configurations are maintained manually, but it is also essential for automatic maintenance and generation of configurations.