CASL

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

Erik Sandewall

Introduction to the Leonardo System and Overview of Major Applications

Incomplete Manuscript – Work in Progress

This project memo pertains to the development of the Leonardo system. Identified as PM-leonardo-004, it is disseminated through the CAISOR website and has URL `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/004/`

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:	2008-10-06

1 Introduction

The Leonardo computation system ([1]) is an experimental system, in the sense of explorative design, where we study the possibility of an alternative overall organization of standard computer system software: programming language system, operating system, user interfaces, and several others. The topic is a large one, and in other articles we have described the Leonordo approach from the point of view of first principles. The present report is an introduction to the use of the actually implemented software system.

The most basic design in Leonardo is its representation language which is called LDX, for the 'Leonardo Data eXpression' language. This is a textual representation of structured information which is used in three major ways:

- It is used for publication purposes in articles and, first of all, in a textbook in Knowledge Representation that is being written ([2]). A preliminary version of this textbook has been used for an advanced undergraduate course and a graduate course at Linköping University.
- It is used as the primary representation language in the Common Knowledge Library ([3]) which is an open-source resource for facts and knowledge that presently contains close to 70.000 entities each with a number of attributes.
- It is used for all programs and most of the data ([4]) in the Leonardo software system. The implementation of the system itself is organized using LDX, and various application systems that are based on Leonardo also represent their information in LDX.

Before proceeding here, the reader should read Chapter 2 in Part I of the above mentioned textbook ([5]). Chapter 1 and Chapter 3 are also recommended reading before the present report.

Sections 2 and 3 here will introduce general aspects of the Leonardo system; Section 4 and onwards will introduce the major application services that have been implemented on the Leonardo platform.

This report is logically followed by three other reports:

- The Leonardo Kernel and Platform
- The CommonLisp Implementation of Leonardo
- The Leonardo Facility for Management of Information Resources

The first two describe the technical aspects of the Leonardo system; the last one describes the use of the system for the management of the Common Knowledge Library ([6]) which is a website providing open access to a relatively large repository of fact and knowledge modules.

---

[1] The name has occasionally been written as Leonordo.

[2] Erik Sandewall: *Introduction to Knowledge Representation.* Available in the 'Research reports' menu item at: `http://piex.publ.kth.se/reports/`

[3] `http://piex.publ.kth.se/ckl/`

[4] with the obvious exception of special-format data, such as (multi)media files.

[5] The direct link to Part I is: `http://piex.publ.kth.se/reports/krf/001/`

[6] `http://piex.publ.kth.se/ckl/`

The application descriptions in the present report contain examples of runs; these examples have been obtained from runs of the actual system. Some of the output has been removed from the original text in these runs in order to avoid irrelevant detail and improve legibility, but otherwise no editing has been made.

2 From KRE to LDX

Chapter 2 of the textbook "Introduction to Knowledge Representation" defines the syntax for *Knowledge Representation Expressions*, or KRE. This notation forms the basis for the Leonardo Data Expression language, LDX, but there are some minor differences because of the needs of the computer implementation. The present section will describe these differences.

Attribute Names

Instead of the attribute `has-attributes` as used in the textbook, one should use the attribute `attributes` in LDX.

For completeness, the full picture is as follows: each type can have both an `attributes` attribute and a `has-attributes` attribute. The contents of the latter is a subset of the contents of the former. We use `attributes` for specifying what attributes are to be preserved when contents are written to files in the operation of the Leonardo system. We use `has-attributes` in the Common Knowledge Library for specifying which of the attribute values are going to be published on the website; the other attributes are for internal purposes in the system.

Substitution of Special Characters in KRE

One of the goals in the design of the KRE notation was that it should be possible to use it both in textbooks and articles, and in a computer implementation. This is why it is expressed using fixed-width or "typewriter" font, and almost exclusively using characters in the Latin-1 character set. However, KRE contains a few minor exceptions to this principle. In particular, the brackets for maplets are written as e.g. `⟦color red⟧` in KRE. The corresponding LDX notation is `[: color red]`.

Alternative Delimiter for Strings

KRE makes a distinction between symbols and strings, and specifies that strings are enclosed in double quotes. This convention is used in LDX as well, but there is a special problem if one wishes to use a string containing a double-quote character inside it. This is handled using an escape character in most other representation systems, but in LDX we have chosen another method. There is an alternative manifestation of strings as

```
[? this is the string?]
```

here and for the rest of this section, please read `?` *as the paragraph character; I have not found out how to represent it in Latex*, which is equivalent to the standard manifestation,

```
"this is the string"
```

Notice that there must be at least one space after the beginning `[?` and that the (first) space is not part of the string. If there are additional space characters there then they are part of the string. Of course the string is not allowed to contain any occurrence of the two characters `[` and `?` in direct succession. (If these two characters to appear in sequence in the textual representation of the string then they should denote the beginning of a substring, but this is not supported by the implementation at present).

In this way one can represent strings containing a `"` inside them. The standard Leonardo printing function checks every string that it is supposed to write, and if there is a `"` inside the string it produces the manifestation of the string in `[?` format, otherwise in `"` format. (It does not check for `[?` or `?]` inside the string). Notice in particular that since \ is not an escape character in this notation, any occurrence of this character inside the manifestation of a string both in `"` and in `[?` format, just represents itself.

One of the reasons for not using the common practice with \ as the escape character is that I want to be able to write Latex coded text in strings, and then e.g. ä has to be written as

```
{\"a}
```

which would have to be coded as

```
{\\\"a}
```

if the standard convention for escape character were used, which is not very convenient. Moreover, the long-term plan is to represent a range of different formatting conventions using notation such as

```
[?html <em>this is text</em>?]
```

and similarly for `xml`, unicode, and a number of others. The present `[?` convention can then be understood as a special case of the general notation.

Representational Conventions for Entityfiles

In brief: each entityfile consists of a sequence of entity descriptions, and each entity description contains the entity being described, a number of attribute-value assignments for the entity, and a number of property-value assignments. The attribute-value assignments are like in KRE; the properties assign "long strings" to the entity in question. Properties can be used for storing e.g. function definitions, or comments.

In the textual manifestation of the entityfile, entities are separated by a line consisting entirely of dashes or equal signs. The last entity is followed by a line consisting entirely of `o` (small letter o) characters. Usually entities are separated by a line of dashes; equality signs are used between sections within an entityfile. More about sections below.

Within each entity description, the separating line of dashes or equality signs is followed by a header line consisting of exactly two dashes, exactly one space, and then the entity which may be a symbol or the expression for a composite entity. It is followed by a blank line, and then the attribute-value pairs.

The property assignments follow after the attribute-value pairs. Each property begins with a line where the first character is the commercial at immediately followed by the property-tag. like in the following example. The following lines are interpreted as text and constitute the property value, up to the next property-tag line or the end of the entity description.

Here is an example:

```
------------------------------------------------------------
-- hamlet-quotation

[: type famous-quotation]

@English-phrase
To be or not to be, that is the question.

@Swedish-phrase
Att vara eller inte vara, det aer fraagan.

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

One particular kind of property is the *blank-tag* property which is used for an S-expression containing CommonLisp code that is executed when the entityfile is read. A blank-tag property must precede all other properties, i.e. it must immediately follow the attribute-value assignments (although one or more separating blank lines are permitted). The first line of a blank-tag property must have a left parenthesis in its first position; all following lines must either be emtpy, or have a space in the first position.

3 General Leonardo LazyDog

This section contains LazyDog information ([7]).

Checking the installation

Usually the Leonardo system is installed by copying certain directories to the host where you want to run it, and then checking the following:

1. There shall be an `Allegro CommonLisp` system on the machine.
2. The directory structure for the Leonardo individual has a structure like the following:

[7] The Swedish word *lathund* literally means *lazy dog*, but the word is commonly used to designate a brief manual containing the essential instructions for performing a task or operating a device. We use the English word with the same meaning in this report.

```
|-- Leonardo-Residence
|---|--- Leonardo-individual
|---|--- Residmap
      ...
```

Both the Leonardo residence and the Leonardo individual may be named freely, but the name of the individual is chosen when it is generated and can not be changed afterwards since the name is hard-wired into certain places in the individual. The `Residmap` directory must always be called `Residmap`; it contains certain files that come into play when there are several individuals in the residence and they send messages to each other. - Unless otherwise noted, all paths mentioned from here on are given relative to the individual, e.g. relative to `Leonardo-individual` above.

3. There must also be a directory called `C:/Leohost` containing a few files describing the host at hand, for example, where certain standard programs are located (text editor, Latex system if applicable, and so on). This directory typically comes as a part of the "installation package" and the user must move or copy it to the right place and edit its few files manually to tell Leonardo where the programs in question are located on the host at hand.
4. Check the contents of the file called

   `Process/main/Defblock/minileo.bat`

   This is the file for invoking a basic version of the Leonardo system. Check that the second line contains a correct path to the `Allegro` interpreter, and edit it otherwise

As an alternative to storing the Leonardo residence on the hard disk of the computer where it is to run, it is also possible to keep it on a detachable memory such as a USB memory stick. In this way it is possible to activate it alternatingly on several different hosts. The only requirement is that each of the hosts has an Allegro CommonLisp system installed, and that the `LeoHost` directory is present and correctly configured on each of those hosts. Moreover, if the CommonLisp system is located differently in the different hosts then one will have to have several variants of the `minileo.bat` file in order to accomodate the different hosts.

System philosophy

Each Leonardo individual is supposed to be self-contained. Apart from operational context information (`Residmap` and `Leohost` directories) it is supposed to contain all its information inside itself. The way to work with it is to invoke a 'run' (also called 'activation') of the individual, usually by clicking the `minileo.bat` or some other, similar `.bat` file in the `Process/main/Defblock/` directory, and then alternate between giving commands to that run, and doing text editing on Leonardo files (files with the extension `.leo`) within the individual. In particular, typical operations are:

1. To issue commands that change the contents of the datastructures in the run, and then to request the run to re-write Leonardo files in the individual so that the new contentes of the datastructures are preserved

2. Text-editing some of the Leonardo files and then issuing a command to the Leonardo run to re-load those text files, thereby importing the new contents into the datastructure.

It follows that if one does a 'write' of such a file and then a 'load' without having edited the file in-between, then there will not be any change in the information in the run (except for obtaining a new timestamp for the 'latest-written' information of the file).

Command loop

Clicking the `.bat` file starts the run and leaves it in standard Lisp input mode. It is recommended to use the Leonardo command-line format. It is started by typing

```
(lix)
```

to the Lisp interpreter. In command mode, you can choose to type in a command (which must be a Lisp symbol), followed by its argument(s), or type in a non-atomic S-expression which is then sent to standard `eval`.

To return from the command loop to Lisp, type the command `lisp`

The `lix` command loop catches errors that occur during the execution of a command, and reports failure. This is convenient in many cases but it also loses information about the character of the error. If one wishes to obtain an error break and be able to track what has happened one can either go back to the Lisp mode, or invoke the special command that disables the catching of errors.

Caveats: Not all errors are caught by the error-catch facility. If an error occurs when files are open for reading or writing, then it is up to the definitions of the individual commands to close those files correctly. Some of the commands in the system fail to do this.

Methods for defining commands, see below.

Basic commands relating to entityfiles

The term 'entityfile' below is the same as 'Leonardo file' above.

```
loadfil foo
    Loads (or re-loads) the entityfile called foo
writefil foo
    Writes the entityfile called foo
updfil foo
    Does loadfil followed by writefil
curfil foo
    Makes foo the current file
loadf
    Loads the current file
writef
    Writes the current file
updf
    Does loadf followed by writef
```

```
setk fie-kb
    Sets fie-kb to be the current knowledgebase. (Cf below)
crefil foo
    Creates a new file entityfile called foo, and makes it a
    member of the current knowledgebase
loadk fie-kb
    Loads the knowledgebase fie-kb. This consists of:
    1)  Load the file called fie-kb
    2)  Do the loadk operation (recursively) on those knowledgebases
        that are specified to be required by fie-kb, if they have not
        already been loaded
    3)  Do the loadfil operation on those entityfiles in fie-kb that
        are specified to be 'mustload' files.
    The requires and mustload information is specified in the file
    called fie-kb.
crek fum-kb
    Creates a new knowledgebase called fum-kb, but does not complete
    the operation
crekk
    Complete the preceding crek operation. A new directory is created
    for the knowledgebase, called Fum, and the entityfiles that are
    created for that knowledgebase are placed in that new directory.
    Thus knowledgebases should always have names that end with -kb.
    (Between the two commands one can do other things that e.g.
    obtain another directory name, but this is special-purpose).
```

Notes of caution

If an entity has attributes then there must be a value for the `type` attribute, and the value must in turn be a symbol that has an `attributes` attribute, since this is what determines what attributes will be written *to* the entityfile under the `writefil` or similar operation.

If an entity does not have any of these then it will be suppressed when the file is written. Therefore, it may happen that you prepare an entityfile by text editing, do a `loadfil`, a `writefil`, look at the file, and find that what you just typed in has vanished. This may be either because you forgot to assign values to `type` attributes, or because the value you assigned does not in turn have an `attributes` attribute. The thing to do in such a situation is to type in the missing attributes into the run, e.g. using something along the lines of

```
(setf (get 'myentity 'type) 'mytype)
```

or

```
(setf (get 'mytype 'attributes) '(seq& (attrib-1 attrib-2)))
```

where the `seq&` construct is the Lisp representation of the Leonardo sequence construct.

The `updfil` command should be used with some caution: if the load operation fails then the following write operation will be done anyway, which may lead to loss of data. If in doubt about the contents of the file, do a `loadf` operation first and then a `writef` operation if the `loadf` went right.

Editing Attribute Values

There are in principle three ways of changing the values of attributes:

- Edit the text file for the entityfile containing the entity in question, and re-load it.
- Use `lix` commands for assigning values to attributes.
- Use the `edo` command which invokes a text editor and pre-loads it with the LDX representation for the particular entity whose attribute is going to be changed. Edit the attribute(s) and exit the editor; the Leonardo system will read back the amended entity description.

If the first or the third method is used then one must of course remember to re-write the entityfile containing the entity before the session ends, otherwise the edit will be lost. In particular, the `edo` operation does not save the entityfile automatically.

The `edo` method is particularly convenient when one the entity is in a very large entityfile which it may take a lot of time to reload. The commands pertaining to this are as follows:

```
curo foo
    Sets the entity foo to be the current 'object' (should have
    been called cure for current entity)

edo
    Invokes the user's preferred text editor on a temporary file
    consisting of a preamble and then the entity-description
    (attributes, etc) for the current object. The user can edit
    this file, save the file, and kill the instance of the
    editor, which returns control to the Leonardo runs where
    the auxiliary file is read back, resulting in update
    accordingly of the current object's attributes, properties,
    and/or definition.
```

Definitions

The blank property or 'definition' part of entities should be executable Lisp S-expressions. They will often be a `defun` expression, a `setf` or `setq` expression, or the like. In addition there is the option of expressions of the form

```
(leodef function-name command-name (arg1 arg2 ...) form)
```

which has essentially the effect of doing both of the following:

1. `(defun function-name (arg1 arg2 ...)  form)`
2. Defining `command-name` as a command that takes the indicated arguments and executes the form

If `function-name` or `command-name` is `nil` then the respective variant is suppressed. The arguments given to `form` are supposed to be Leonardo represented things and in command mode the arguments are to be written in Leonardo representation. Thus if I do

```
(leodef fun foo (a) (...))
```

and type the command

```
foo <red green>
```

then this has the same effect as evaluating

```
(fun '(seq& (red green)))
```

this argument being the internal representation of ⟨`red green`⟩.

Moreover, the body in `leodef` definitions is sent through a special macro expander that is implemented within Leonardo (not the standard Common-Lisp macro expander). There are macros that provide a reasonable notation for construction of, and access to Leonardo constructs. I set them up and tried to be consistent about using them but more recently I have been sloppy and done `cadr`-type operations directly on the low level Lisp representation of the Leonardo constructs, unfortunately. (Will have to clean this up).

The `.leos` files

The command `writefil` and similar ones write two versions of an entityfile: a `.leo` version which is the one you usually work with, and a `.leos` version in the `cl` subdirectory of the knowledgebase directory where the `.leo` file lives. The `.leos` file is a plain lisp file so it can be loaded with the lisp function `load` without any preconditions. In principle this has the same effect as loading the `.leo` file, except the `.leos` file loads much faster, but there are a few small but significant differences:

1. When function definitions (`defun`, `leodef`, etc) are read in `.leo` format, the system loads the entire definition as a string, stores it with the entity in question, and also `eval`'s it. This is necessary in order to be able to write the definition to the file in the `writefil` operation. However, loading the `.leos` definition does not store the definition as a string in that way, it merely executes it. Therefore, if one loads an entityfile in its `.leos` version and then does a `writefil` without first doing a `loadfil`, then all the definitions are lost, in both the `.leo` and the `.leos` file.
2. A few files, such as `Core/bootfuns.leo`, contain properties with the property-name `Exec-leos`. These are merely properties from the `.leo` point of view, but they are placed as evaluable in `.leos` files. They are used for definitions that are to be run at an early stage during the system startup process, where `.leos` files are loaded and not `.leo` files. In this way one can safely work with these definitions, if one wishes to change them, by just loading and writing `.leo` files in the standard fashion, and they take effect during startup when `.leos` files are loaded.

Notice also that the `loadk` operation for knowledgebases loads files in `.leos` format.

4 Document Preparation and Management

The first practical use of the Leonardo system by the present author was for administrating the collection of articles and reports, including both those that had been written before and those that were in the manuscript stage. The application in question is called `Madman`, for *Management of Articles, Documents, Manuscripts And Notes*.

Processing of Documents

Each document (including articles, etc) that is managed is assumed to have its own directory in the file system. Each document is also represented by its own entity in the sense of Leonardo. The way I use this myself is that document entities are formed like e.g. `(doc 2008 pm-003)` referring to manuscript number 3 during 2008. The symbolic function `doc` (which actually has another name in reality) is implemented so that it computes the directory of the document in question and assigns it as an attribute of this composite entity.

The entity for the document has attributes for the obvious metadata, such as author, title, and so forth. The files in the document's directory use a set of systematic naming conventions. For those documents being prepared using Latex, which is the case for most of them, there are a few standard files such as:

```
paper.tex
descr.tex
body.tex
```

Here, `body.tex` contains the bulk of the contents of the article, and `descr.tex` contains the metadata but in Latex format. In principle it should be possible to generate `descr.tex` automatically from the metadata in LDX, but this has not been implemented yet. The file `paper.tex` is a top-level file that imports the relevant macro definition files as well as `descr.tex` and `body.tex`.

In most cases I prepare the manuscript using my own markup language which I find more convenient than the standard Latex notation. Therefore, the real source file is `body.aml`, and the following commands are used in sequence for the preparation of a document in Latex:

```
gal      Use body.aml to generate body.tex
lam      Run Latex on paper.tex
dpdf     Generate the pdf file from the Latex output
```

Alternatively, it is possible to use `body.tex` as the source file and to hand-edit it directly. In this case the `gal` command is not used.

There are also a number of other commands, including:

```
galp     Do gal, lam, and dpdf in succession
dps      Generate a postscript file from the Latex output
bim      Run bibtex
asy      Run the Asymptote program (more below)
```

These commands are defined without an explicit argument. They operate on the 'current document' which is set by a command such as

```
curdoc (doc 2008 pm-003)
```

There are also commands for registering a resource in a catalog or other group, etc.

It is often desired to format the same document using several different style files. In principle this shall only affect the `paper.tex` file and not the contents of the `body` files. One may either edit `paper.tex` or set up several alternative files that are like `paper.tex` but with different contents with respect to paper style. In the latter case one has to change the top-level name away from the default which is `paper`, using for example

```
curtop ecai-paper
```

References

The user's catalog of metadata for articles that he will sometimes wish to cite can be maintained either as Bibtex files in the standard way, or as LDX entities in which case the Bibtex information is generated for each article.

Figures

The user can of course handle figures the way he likes within Latex. However, we have also implemented a facility that is particularly useful for the kind of "boxes and arrows" diagrams that often occur in the kinds of articles we write. In this facility, each diagram is represented as an LDX entity with a property for a high-level description of a diagram. For example, the following is the definition of a diagram for the Nixon diamond:

```
-----------------------------------------------------------
-- nixonfig

[: type asyfigure]
[: caption "The Nixon Diamond"]
[: vsize 140]
[: nullvalued {def latest-rearchived}]

@Asy-script
((-20 caption)
 (Nixon box1 0 10)
 (Republican box1 70 50)
 (Quaker box1 -70 50)
 (Pacifist box1 -70 90)
 (NonPacifist box1 70 90)
 (t horiz-neg Pacifist NonPacifist)
 (t vert Republican NonPacifist)
 (t vert Quaker Pacifist)
 (t vert Nixon Republican)
 (t vert Nixon Quaker)
     )
-----------------------------------------------------------
```

The `Asy-script` property specifies for example that there shall be a box containing the word `Quaker` in position (-70, 50). It also specifies that there shall be a vertical arrow from the box labelled `Quaker` to the box labelled `Pacifist`, and that there shall be a field in vertical position -20 containing the figure's caption as specified in the `caption` attribute.

The command

```
asy nixonfig
```

converts this script to `Asymptote` notation, invokes the Asymptote program on it, and displays the result. As a result the figure can be included in the manuscript, e.g. using a command in the `.adl` sourcefile.

It should be clear from this brief description and from the example that the document-support facility is command-oriented and procedural rather than graphical. This has its pros and cons: the immediate convenience of a graphical, wysiwyg interface can be balanced against the ease of adding higher-level facilities in a command-oriented approach. In our case the latter aspect is more important, in particular because in the long term we want to consider the document-processing commands as *actions* that can be used in a context of planning, plan execution and high-level autonomy.

5 Management of the CKL Knowledgebase

The Common Knowledge Library (CKL) ([8]) is a service offering open access to a library of almost 70.000 entities, most of them in the areas of world geography and in scientific publication (journals, publishers, etc). The management and gradual extension of the CKL requires support services of the following kinds, which have been implemented based on the Leonardo platform:

- Import of information from existing sources (web pages, databases, etc)
- Checking and correction of sources in a combination of automatic and manual operations
- Management of IPR information concerning acquired information and CKL contents
- Generation of presentation versions of information files

These services are further described in a separate memo, *The Leonardo Facility for Management of Information Resources*. One of the significant aspects of this facility is that it provides a context for introducing an ontology for common-sense and other real-world information.

6 Communicating Individuals

The following example demonstrates the use of multiple Leonardo individuals that communicate by passing messages to each other. It also illustrates the representation of on-going events as well as past events using LDX.

[8] `http://piex.publ.kth.se/ckl/`

The Example

Consider the following method description in LDX notation:

```
------------------------------------------------------------
-- method4

[: type method]
[: plan {[intend: t1 t2 (remex: lar-001-004 (query: whatyourbid))]
         [intend: t1 t3 (query: whatbid)]
         [intend: t4 t5 (query: whatproposal)]}]
[: time-constraints {[afterall: {t2 t3} t4]}]
------------------------------------------------------------
```

This is a plan, i.e. a kind of high-level procedure, for performing the action `query:` three times with three 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 agent (called `lar-001-003` in our specific run of the plan) but the first mentioned action is to be remote executed (therefore `remex:`) in another agent called `lar-001-004`.

The representation that is used for expressing the plan is general-purpose in the sense that it is used for both procedures and data, as well as for information with an intermediate status such as the 'plan' above. It is organized in terms of *entities* each of which has a number of *attributes*. In our example, there is an entity called `method4` with three attributes `type`, `plan`, and `time-constraints`. All entities must have a `type` attribute, and the value of this attribute determines what other attributes may be present.

The example is a straightforward use of the LDX notation. Notice that `(query: whatbid)` is a composite entity that has a type and attributes, just like the atomic entity `method4`.

Consider now 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-001-003` and `lar-001-004` which may be located on the same computer or on two different ones. The interactions on `lar-001-003` go as follows, after the obvious startup of the system:

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

067-> selmeth method4
```

Each interaction is numbered; 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. In our demo we have shortcut this by 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':

```
068-> seg
    > (adogoal: 66 (achieve: a b c))

069=>
----> What is your bid? 16200

070-> Continuing:
AI (b: 68 (query: whatbid)) completes:
Succeed, result: 16200

071-> Continuing:
Received outcome for action started at: 68

072=>
----> What is your proposal? 13000
073-> Continuing:
AI (b: 71 (query: whatproposal)) completes:
Succeed, result: 13000
Completed goal: (achieve: a b c) adopted at: 66
```

The following is what happens. When the user types in `seg`, the action `(query: whatbid)` starts to execute in the individual at hand, which has the effect of displaying the prompt `What is 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 the prompt `What else do you want to say?` on its screen. 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 `whatbid`:

```
------------------------------------------------------------
-- whatbid

[: type output-phrase]
[: englishphrase "What is your bid?"]
[: swedishphrase "Vad r Ditt bud?"]
------------------------------------------------------------
```

The user for the first individual answers the prompt with the value `16200`, which counts as interaction `069`, and the system confirms completion of that action in interaction `070`. The first individual also receives the value from completion of the action in the other individual, in interaction `071`. 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-001-003` to start performing the third action in the plan, leading to interaction `072`, after which the goal is reported as completed in interaction `073`.

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:

```
076-> log
(adogoal: 66 (achieve: a b c))
  Plan name:  method4
  Plan:  {[intend: t1 t2 (remex: internal-ch-02 (query: whatelse))
                   :done t]
          [intend: t1 t3 (query: whatbid) :done t]
          [intend: t4 t5 (query: whatproposal) :done t] }
```

This is like above, except that each of the actions has been marked as completed. The format of the logs is a slightly sugared variant of LDX. The command `loa`, for 'list old actions' displays the actions that were performed in the first individual, as follows:

```
075-> loa
  (b: 68 (query: whatbid))
      Towards goal  (adogoal: 66 (achieve: a b c))
      State         [result: 16200]
      Subactions    <(b: 69 (ask: whatbid))>
      Outcome       [result: 16200]
      Endtime       71
  (b: 68 (remex: lar-001-004 (query: whatelse)))
      Towards goal  (adogoal: 66 (achieve: a b c))
      State         [requested:]
      Subactions    <>
      Outcome       [result: 12900]
      Endtime       72
  (b: 69 (ask: whatbid))
      Subaction-of  (b: 68 (query: whatbid))
      Outcome       [result: 16200]
      Endtime       70
  (b: 71 (query: whatproposal))
      Towards goal  (adogoal: 66 (achieve: a b c))
      State         [result: 13000]
      Subactions    <(b: 72 (ask: whatproposal))>
      Outcome       [result: 13000]
      Endtime       74
  (b: 72 (ask: whatproposal))
      Subaction-of  (b: 71 (query: whatproposal))
      Outcome       [result: 13000]
      Endtime       73
```

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.

The examples of logs above have been slightly edited so as to remove output that was irrelevant for the example. We have also edited the first argument of the `remex:` operation so that it appears to refer to the individual where the remote execution is performed. In the current system the first argument shall be the name of the channel through which the communication takes place, and where the target individual is at the other end. It is a trivial modification of the software to change this if desired.

The present implementation is set up so that each individual only listens to (at most) one channel and communicates with one other individual. Allowing multiple channels is a small change which does not require any change of the software structure.

7 Discussion

The command-line dialog with Leonardo is reminiscent of what one has in the shell of an operating system and in an incremental programming language in the Lisp/Perl/Python tradition. The use of actions and plans, and the explicit representation of plans and their execution as data structures, is reminiscent of what one has in 'intelligent agent' systems and in knowledge-based systems in artificial intelligence. However, one can also make comparisons in other directions, in particular with discrete-event simulation systems and with dialog management systems.

Consider, for example, the software for the robotic dialog environment (RDE) that was developed in the WITAS project (references). The task there was to develop at system for spoken dialog between a human operator and an unmanned aerial vehicle (UAV), which in our case was a helicopter.

This required the use of a *dialog engine*, that is, a piece of generic software that administrates a multi-threaded dialog and coordinating auxiliary systems such as a speech analyzer, a speech generator, a complementary graphic interface, and of course the communication with the actual UAV. The project as such also required the development of an environment simulator that the dialog system could be tested against. Looking back at the structure of the dialog manager and the simulator, it is clear that both of them require exactly the facilities that the Leonardo system offers, in particular, the use of a knowledge-base for the models of the environment and the dialog, the explicit representation of goals and actions, and the use of interaction and of distributed computation.

Our main argument is not that our system can be used for just about everything, however; our argument is that so many different things are maybe not needed. It is an argument about the overall structure of computer software today, which we are used to thinking about in terms of operating system, programming language, and many other kinds of software artifacts. There are a number of designs that recur throughout these softwares: the use of typed objects and the descriptions of types, the use of 'functions' or 'procedures' with their arguments, and so forth. These occur repeatedly in operating system shells, in programming languages, in database systems, in communication frameworks such as CORBA, in markup systems such as LaTeX, in the specification of dynamic columns and fields for spreadsheet systems, in ontologies as defined in OWL and in XML, in webpage specification languages such as PHP and JavaScript (why should there be different languages in the web server and the browser; both are used for producing effects in the web browser), and so on ad nauseam.

This *standard software architecture* is very strongly entrenched due to the gigantic investments that have been made both in software and in training for it. It is also strongly entrenched on the academic side, where the field of computer science is perceived as *consisting* of subfields for each one of several of those types of artifacts. We have subcommunities for operating systems, programming languages and systems, database systems, and so forth. But this is also very costly: this baroque software architecture costs a lot to maintain and extend, and training for it requires people to learn about a lot of things that differ only trivially.

The working hypothesis of the Leonardo project is that it is possible to do things in a better way, and to develop an architecture for the overall system where these kinds of duplication do not occur. The LDX representation is the system's cornerstone. The kernel of the implemented system consists of programs for conversion in both directions between the textual reprsentation and the data-structure representation of LDX, together with a minimal command executive, a bootstrap system, a system for duplication and mobility of individuals, and routines for administrating groups of entities on several levels.

The facility for goal-directed execution that was shown in one of the examples is not properly part of the kernel, but it is built directly on the kernel and it is expected that many applications will prefer to use its command executive rather than the simple one that comes with the kernel.