CASL

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

Erik Sandewall

How to Begin Using the Leonardo System

For Remus Version ap-0030

This project memo pertains to the development of the Leonardo system. Identified as PM-leonardo-007, it is disseminated through the CAISOR website and has URL `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/007/`

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:	2009-05-27

1 Introduction

This report describes the central parts of the Leonardo system from an operative point of view. A suite of simple examples illustrate what one can do in this system, and how it is done. The report has been written for students and colleagues in our department who have read some of the other articles and reports about the Leonardo system [1] and that want to give it a try, but also for those that simply want to get a feel for what the system does without necessarily wanting to run it themselves.

It is possible to try out the Leonardo system since reasonably stable versions of a small Leonardo system, called Leordo, now exist for use on Windows and Linux systems.

One of the first things that you will notice when running a Leonardo system is that programs and data are integrated. In a conventional programming language you write a program, and then you apply that program to your data. In Leonardo you have instead one single structure that is called an *individual* that contains both programs and data. In fact, it also contains various other kinds of information that are intermediate between programs and data. Everything you do during a Leonardo session (once the system is up and running) uses and modifies the contents of that individual.

section 1 of this report describes how you can get your own Leordo individual to work with, and section 2 describes some practical handling issues for such an individual. The reader who only wishes to read about the system may wish to proceed directly to section 3, therefore.

1.1 Choice between Server and Personal System

Your first choice is whether you wish to run Leonardo on a server or on your own personal computer.

There is one resource for the server option at present, namely a computer called `elektrika` which you access as IP 130.236.69.194 (no symbolic address yet). It is a linux machine which one has to access using an `ssh` connection.

In the personal computer case, you must arrange to have a computer where Allegro CommonLisp (ACL) is installed. If this is a department-owned computer then you can install the 'professional' ACL system under IDA's site license. Please contact me for details. If it is your own computer, or someone else's besides IDA, then the best solution is to install the free version of Allegro's system, called Allegro Express. Instructions for this follow below. Both the commercial version and the express version exist for both Windows, Linux, and Mac.

Leonardo individuals do not contain any machine-executable code, which means that they need an implementation of a programming language, called the *host language* in order to be operational. They are organized so that it shall be possible to run one and the same individual with several host languages, but at present this has only been implemented for one lan-

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

guage, namely CommonLisp, and in particular Allegro CommonLisp. [2] The best languages for implementing Leonardo are the incremental ones, so e.g. Python would also be a good choice for implementation language.

1.2 Choice between ANSI Base and Modern Base Lisp

When choosing and installing a Lisp system and a corresponding Leonardo individual, you should be aware of the distinction between ANSI Base and Modern Base Lisp systems. The former are designed in such a way that some aspects of their input are automatically converted to upper-case characters: you type lower case and you prepare your input files in lower case, but they are converted to upper case when read and appear in upper case when results come out again. Modern Base systems do not do this; they preserve both upper and lower case.

Case-insensitive operation was the historical practice in Lisp and it became entrenched through the ANSI Standard. ACL offers both possibilities, but most other implementations of Lisp offer only ANSI Base.

Leonardo has been implemented in a context of Modern Base, but in such a way that it is possible to convert a Modern Base Leonardo individual to one that uses ANSI Base. The conversion in the opposite direction is more tricky and has not been implemented. However, the ANSI Base variant of Leonardo is not fully tested yet, and it is strongly recommended to only use the Modern Base option unless you want to experiment with running on other Lisp systems besides ACL.

1.3 Obtaining an Allegro Express System for Leonardo

If you are a student and want to run Leonardo on your own PC or desktop then probably you want to get the free version of ACL. (There is also a "student version" at a lower price than the professional one, but I do not know anything about it). To obtain the Express system visit Franz (the company producing the Allegro system) at

`http://www.franz.com/`

click "Downloads", then click "Allegro CL Free Express Edition", then "Download now ..." where you get to identify yourself, and then download according to your choice of operating system. The download is an executable file that you execute to install (Windows) or unzip as usual (Linux et al).

However you must make one more step, which is because the variant of the system that you get by the Express option has two features that one should get rid of: it is an ANSI Base system, and it comes with an attached editor that does not really fit into the Leonardo way of doing things. Fortunately, it is able to convert itself to a Modern Base system. Do as follows, once the system has installed itself.

[2] We are working on providing e.g. LispWorks as an alternative, but this is not entirely trivial since the present implementation uses a few auxiliary packages in Allegro.

On a Windows system: Position yourself where the system is located, for example under

```
C:\Program Files\acl81-express\
```

Click the file `allegro-express.exe` in this directory so as to start the run. This will open a group of two windows. Delete the editing window (the one to the right). In the window called the debugging window, type in or paste in the following expressions and hit the Return key after each of them:

```
(build-lisp-image "mlisp.dxl" :case-mode :case-sensitive-lower
                    :include-ide nil :restart-app-function nil
                    :restart-init-function nil)
```

and

```
(sys:copy-file "sys:allegro-express.exe" "sys:mlisp.exe"))
```

Keep an eye on the contents of the `acl81-express` directory as you do this. The effect of the first operation should be the addition of a file called `mlisp.dxl` in that directory; the effect of the second operation should be the addition of a file `mlisp.exe`.

If the second operation fails and `mlisp.exe` does not show up, then it is also possible to simply drag-and-drop `allegro-express.exe` to a copy that you rename `mlisp.exe`.

After this, click the newly created file `mlisp.exe`. If everything has worked out right, this should open one single window that just shows a simple greeting. Check that it is correct by typing a single (lower-case) `t` into it. If it is correct then it shall answer you with `t` but if you have accidentally obtained an ANSI Base system then it will answer you with `T` instead. In this case, try the above procedure again.

On a Linux system: The same steps as for Windows should apply, but I have not yet checked them out. Please try to do it by analogy with the above, and report if you have any problems.

1.4 Downloading Leonardo for Use on PC

In personal computer mode, once the ACL system is in place, you can proceed to installing a copy of the Leonardo system. Do the following: identify the place for download on the Leordo website [3] It offers two files to download, called `leordo-remus.bat` (a small file) and `leordo-remus.txt` (around 1100 KB). Both of these are text files so they are easy to work with. Download them and place them in the same directory, for example `E:/Leordo/`. Edit the `bat` file and make sure that the path to the ACL system on its first line is correct. (Do *not* remove the semicolon at the beginning of the first line.) Then, invoke the `bat` file by clicking it.

This will start an installation run that uses the ACL system and that splits up the contents of the accompanying `txt` file into smaller files that are organized in subdirectories called `remus` and `Residmap` under the chosen directory. The former one is your Leonardo individual, the other one contains a few auxiliary files that are intended to be shared between individuals if you have additional ones besides `remus` in the same directory.

[3] `http://www.ida.liu.se/ext/leordo/`

When the installation run is complete, then descend into the `remus` individual to the subdirectory `.../remus/Process/main/`. It contains a file called `minileo.bat`. This is the one to invoke in order to start a session with your copy of the `remus` individual.

1.5 Using Leonardo on the Elektrika Server

We switch now to the case where you want to run Leonardo on the Elektrika server where it has already been installed. In this case you do not have to worry about downloading and installing ACL since it has already been done.

The server is accessed using ssh, so in order to use this option you must have a program such as SSH Secure Shell (available within IDA) or Putty (freeware).

You must also have an account on Elektrika. Contact me in order to get it.

Finally, you must have your own Leonardo individual. This is because there are several users on the server, and because when you work with Leonardo you change the contents of the Leonardo individual you are working with. Therefore you need your own.

This works as follows. When I set up your account, I also create a new Leonardo individual for you. Probably it will be called something like `rom-7` with some number (e.g. 7) to distinguish it from its siblings. The original files for this individual will be located at

```
/home/erisa/leonardo/rom-7/
```

However, in order to use it you must first copy it to your own directory. If your login is `mylog` then copy the entire `rom-7` directory so that you have e.g.

```
/home/mylog/leordo/rom-7/
```

Like in the case of Windows, the surrounding directory which is `leordo` in the example can be named and placed arbitrarily, but the individual (e.g. `rom-7`) and its contents should not be renamed or rearranged.

The `/home/erisa/leonardo/` directory contains other individuals as well, but you do not need to copy them. However, you must make a symbolic link (a kind of virtual copy) of the individual called `romulus`, typically using the following shell command

```
ln -s /home/erisa/leonardo/romulus ~/leordo/romulus
```

In this way when you inspect your `leordo` directory it looks like there are two individuals in it, namely, `romulus` and `rom-7`. The reason for this arrangement is that while `rom-7` is your own individual that you will be working with, it has very little contents at least at the outset, and when you run a session with it it will load entityfiles (i.e. files containing programs and data) from the `romulus` individual which is its virtual neighbor. However, it is only able to read from it and can not write to it, according to the file protections under Linux. More about this below.

After these preparations you should be able to start a session with your Leonardo individual `rom-7`, or whatever its name is. To this end, position yourself at

```
/home/mylog/leordo/rom-7/Process/main/
```

and invoke

```
./linuleo.lex
```

This should start the session.

2 Introductory Use

This section describes how to use the command-line interpreter together with a text editor, and how to privatize your Leonardo individual so that you can then start adding information to it.

2.1 Entering the Command Line Interpreter

When you start a session using either of the two approaches that were described above, you will be in the top loop (also called the read-eval-print loop) of the ACL CommonLisp system. This means that the system expects you to type in a variable or a parenthesized expression (an S-expression) that it can evaluate and execute, and it will show you the value. For example, if you type in

```
(+ 4 5 6)
```

it will show the value 15. However, Leonardo has its own command loop which is started by typing `(lux)` to the CommonLisp.

The format of commands to the `lux` command loop is described in the next section. If an error occurs then you are taken back to the ACL top loop, and it is recommended to do

```
:reset
(lux)
```

in order to get back into operation. Your work has not been lost, normally. If you are in the `lux` command loop and want to get back to Lisp in an orderly fashion, type `lisp`.

In order to terminate the entire session, type `exit` to `lux` or `(exit)` to the ACL top loop.

If you already know Lisp then there are many other things you can do on the Lisp level or when an error occurs, but the above is what you need in order to use Leonardo without bothering about Lisp.

2.2 Session and Editor

The *persistent manifestation* of your Leonardo individual is a collection of files in the individual's directory, in the sense that this is how the individual is preserved between sessions. During a session, there is both the persistent manifestation in those files, and the *dynamic manifestation* as data structures in the run, and there are frequent exchanges of information between those two manifestations. In order to follow what happens it is therefore

recommended that you watch the contents of your individual's directories at the same time as you type commands to the session.

If you are using the server installation on Elektrika it is recommended to use two windows by running the ssh program twice with the same login, and to use one window for seeing and operating on the directory and for editing Leonardo files while the session goes in the other window.

The first level of subdirectories inside an individual contains subdirectories of two kinds: *knowledgeblocks* and *special purpose subdirectories*. The following are the special purpose subdirectories:

```
Ontology
Process
Savestate
Starter
```

All others subdirectories in the distributed system are used for knowledgeblocks and consist of *entityfiles*. An entityfile is a collection of *entity descriptions* that are expressed in Leonardo notation.

If you are using the server approach you will notice that your individual has very little information in its knowledgeblocks, but the `romulus` individual that you linked to contains much more. Those knowledgeblocks can therefore be read, but not written, by your individual. The special purpose subdirectories are always local to each individual and contain information that is special to it.

You will regularly want to look at the contents of entityfiles and to edit them. Therefore you should make a choice of text editor that you want to use for this purpose, for example `emacs` in the case of Linux. Any text editor will do and at present there is no special link between Leonardo and a text editor.

2.3 Available Knowledgeblocks

The following knowledgeblocks are included in the Leordo systems `romulus` and `remus`

```
core-kb
els-kb
exec-kb
exformat-kb
indiv-kb
modstruc-kb
sysman-kb
text-kb
```

They are located in subdirectories called `Core`, `Els`, and so forth. The knowledgeblock `core-kb` is always loaded at the start of a session; the others can be loaded using the `loadk` verb, for example as

```
loadk sysman-kb
```

Each knowledgeblock contains information about which other knowledgeblock it *requires*, and the operation of loading a knowledgeblock always

first loads the other blocks that it requires and that have not already been loaded. This proceeds recursively. In particular, the operation

```
loadk indiv-kb
```

also loads all the other blocks mentioned above except `sysman-kb`. (At present `indiv-kb` is not available in the `romulus` individual on the Elektrika server).

2.4 Privatizing Catalogs

In the case of the server version of Leordo, each new individual must *privatize catalogs* in order to make it possible to define additional knowledgeblocks and entityfiles that are specific to the individual in question. Before privatization, the individual merely loads entityfiles from neighboring individuals, and they are read-only.

In the case of the PC version this is not an immediate issue since the distributed individual has already its private copies of all the above mentioned knowledgeblocks. However, the following information about how the catalogs work may be of use even for users of the PC version.

A *catalog file* is an entityfile that contains information about where other entityfiles are located. It is somewhat analogous to the collection of paths in an operating system: given the name of an entityfile it specifies the path to where that file is located in the directory structure.

Leonardo catalog files operate in several steps. The root of the catalog structure is an entityfile called `self-kb` that is local to each individual and that is rarely changed. It specifies the location of a few files including `kb-catal` which in turn specifies the location of the catalog files of the knowledgeblocks. Each knowledgeblock, for example `core-kb` is represented by a catalog file containing the paths to all the entityfiles in that block except itself, and `kb-catal` contains the paths to files such as `core-kb`.

When a new Leonardo individual is created, it has its own `self-kb` entityfile which specifies that `kb-catal` is located in the parent individual. In the Linux case above, since `rom-7` was created from `romulus`, the `self-kb` of `rom-7` will initially specify the path for `kb-catal` as

```
/home/erisa/leonardo/romulus/Process/main/Defblock/kb-catal
```

This `kb-catal` file maps the names of knowledgeblock files to other paths in the structure of `romulus` or its parents in ascending order. This means in turn that it is not possible for `rom-7` to modify its entityfiles at all, since they are all located outside its own structure and nothing can be changed.

Therefore, in order to add contents to `rom-7`, the first step must be for it to obtain its own copy of `kb-catal`. This is done using the following command

```
imp-kbc
```

which has the effect of changing `self-kb` so that it will specify the following path for `kb-catal`

```
/home/mylog/leordo/rom-7/Process/main/Defblock/kb-catal
```

Technical note: this has shown the expanded paths; actually the system uses relative paths so that the paths being stored are

```
../../../romulus/Process/main/Defblock/kb-catal
```

and

```
../../../rom-7/Process/main/Defblock/kb-catal
```

respectively. All relative paths are defined on the basis of the directory

```
... /Process/main/
```

which is where sessions are invoked.

After this, it is possible to add more knowledgeblocks, since the path information for the new block is located in `kb-catal` which is now local to `rom-7`. In order to add more entityfiles to an existing knowledgeblock, for example `sysman-kb` one must first import it in the same manner as for `kb-catal`. The following command

```
impkf sysman-kb
```

obtains a copy of the entire knowledgeblock given as the argument for use by the individual itself, and updates the access path in `kb-catal` accordingly.

In the case of server mode use, newly bred (created) individuals on Elektrika have not had any of this done to them, so before doing anything else with your individual you should execute the following commands in order

```
loadk sysman-kb
imp-kbc
```

The `loadk` command loads the knowledgeblock given as argument into the current session, and it must be performed first since `sysman-kb` contains the definition of the `imp-kbc` command.

In the case of Windows mode, these knowledgeblocks have already been imported in the distribution individual `remus` so you do not have to worry about it. However, it is useful to know about this structure when you create new knowledgeblocks and entityfiles.

3 First Exercises

This section describes the most elementary operations in Leonardo. These operations are basic for almost everything that follows. In order not to get stuck because some definition is missing it is best to load several of the knowledgeblocks at once when the session starts, using the following sequence on PC systems:

```
loadk indiv-kb
```

or the following sequence on the Elektrika server:

```
loadk els-kb
loadfil els-demo
loadk exformat-kb
loadk exec-kb
```

Among the others, `text-kb` and `modstruc-kb` are loaded automatically since they are listed as prerequisites of the above, `core-kb` is loaded al-

ready in the startup sequence, and `sysman-kb` is not needed for any of the examples.

3.1 Creating a Simple Knowledgeblock

As an example, let us create a simple knowledgeblock with some minimal contents. We give the following commands in sequence to the `lux` command loop.

```
crek groceries-kb
setk groceries-kb
crefil fruits
```

The first command has the effect of creating a new knowledgeblock, called `groceries-kb`. It is manifested as a directory called `Groceries` containing a file called `groceries-kb.leo`, e.g.

```
E:\Leordo\remus\Groceries\groceries-kb.leo
```

or

```
/home/mylog/leordo/Groceries/groceries-kb.leo
```

The entityfile `groceries-kb` is used for keeping information about additional entityfiles that will be added to the same knowledgeblock.

The `setk` command above makes `groceries-kb` the current knowledgeblock, for use by the command `crefil`.

The `crefil` command creates an entityfile called `fruits` in the current knowledgeblock, e.g.

```
E:\Leordo\remus\Groceries\fruits.leo
```

All of this is done through commands within the Leonardo session.

Next, you want to add information to the entityfile for fruits. This can be done either through commands to the Leonardo session, or by text-editing the file called `fruits.leo`. Consider first the text-editing option. Apply the text editor to the file `fruits.leo`. Its initial contents is a few lines specifying the attribute for the *entity* `fruits`, followed by a line of the form `oooooooo` but longer.

Now add the following information immediately before that final line.

```
------------------------
-- fruit

[: type Type]
[: attributes <has-colors plant-type>]

-------------------------------
-- mandarin

[: type fruit]
[: has-colors {orange}]
[: plant-type tree]

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

```
-- grape

[: type fruit]
[: has-colors {green blue yellow}]
[: plant-type bush]
```

This of course defines two entities, called `mandarin` and `grape` both of which have the type `fruit`, preceded by a specification of that type. The exact number of dashes in the dashed lines separating the entities is not important.

After having saved the text-edited file, go back to the Leonardo session and do

```
loadfil fruits
```

This loads the contents of the entityfile into the session, where you can perform operations that use this information, or update it. After updates, the operation

```
writefil fruits
```

re-writes the file using the current attribute values of the entities in the file.

The load and write cycle is a bit sensitive and it is strongly recommended to have backup copies of files. For example, if you should have spelled the type alternatingly as `fruit` and as `fruits` in the example, so that the specified type of some entity is undefined in the session, then information will be lost in the `writefil` operation since there are no known attributes for what has been specified as the type of an entity. Therefore, take backup copies of files when you text-edit, and double-check that the files generated by `writefil` have the full and intended contents.

3.2 Operations on Entities

This section describes a few of the operations that can be done on entities and entityfiles during a session. We begin with an example. Suppose the entityfile `fruits` has been initialized in the way shown above, and the following operations are performed in the session. Each operation is on one line, and followed by the response from the Leonardo session.

```
039) .(get mandarin has-colors)
{orange}

040) put mandarin plant-type small-tree

041) addmember (get mandarin has-colors) yellow

042) .(get mandarin has-colors)
{orange yellow}

043) writefil fruits
nil

044) .(get fruits contents)
<fruits fruit mandarin grape>
```

Each command line consists of a *verb* followed by an appropriate sequence of *arguments* and/or *parameters*. The examples above only involve arguments and no parameters; parameters are different in that they consist of a *tag* and a corresponding *value*. The full stop (.) can be used as a verb whose only effect is to show the value of its single argument.

In general, arguments may be symbols, strings, numbers, sets, or sequences, as shown in these examples, and in all cases they just represent themselves. In addition, arguments can be given as *forms*, for example `(get mandarin has-colors)`; the value of `(get` *e a*`)` is obtained as the *a* attribute of the entity *e*. Forms can occur inside forms.

The verb `put` takes three arguments and assigns the value of its third argument to the entity-attribute combination of its first two arguments.

The verb `addmember` assumes that the value of its first argument is a set or a sequence, and adds the value of its second argument in the obvious way.

Interaction number 44 shows how system information can be accessed in the same way. The name of the entityfile containing this information is `fruits`, and this entity has a `contents` attribute with the sequence of the entities belonging to the entityfile, beginning with itself. This list is used for deciding what contents are to be included in an entityfile when it is generated. For example, if one does the following sequence of commands

```
put blueberry type fruit
put blueberry has-colors {blue}
put blueberry plant-type bush
addmember (get fruits contents) blueberry
writefil fruits
```

then looking at the file `fruits.leo` you will notice that towards the end it will contain

```
------------------------------------------------------------
-- blueberry

[: type fruit]
[: has-colors {blue}]
[: plant-type bush]
```

This is an example of how the contents of entityfiles can be changed not only by direct text-editing, but also by commands that are given in the Leonardo session.

3.3 Defining Scripts

Always using the elementary verbs in Leonardo would not be very efficient. The present section describes how to define higher-level verbs from the given, elementary ones.

Suppose we want to define a command that will perform all the assignments for a kind of fruit, so that instead of the first four commands in the blueberry example we would just write

```
newfruit :name blueberry :color blue :plant bush
```

This definition must then be written as a separate entity in an entityfile, for example as follows:

```
-------------------------
-- newfruit

[: type leoverb]

@Performdef
[soact [put .name type fruit]
       [put .name has-colors {.color}]
       [put .name plant-type .plant]
       [addmember (get fruits contents) .name]]
```

The example shows how the successive commands are grouped together using the operator `soact` which stands for "sequence of actions." It also shows how the parameters of the invoking action are referred to by preceding their tag with a stop character instead of the colon.

In fact, some versions of Leonardo require you to write a subexpression such as `(param name)` instead of `.name`. The former notation is always accepted, and the latter one has the status of a convenient abbreviation.

Scripts can invoke other scripts in the obvious ways and using "call by value," so that the parameters in an action record are first evaluated and bound to their respective tags, and then the script for the verb in the action record is executed in the context of those bindings.

In order to make a script such as this one available to a Leonardo session, it has to be written into an entityfile like in the example above and then the entityfile must be loaded into the session using e.g. the `loadfil` command. One additional thing is needed, however: the leading entity for an entityfile containing command definitions (the entity `fruits`, for example) must contain the following attribute assignment:

```
[: in-categories {verbfile}]
```

This attribute is used by a handle on the file loading operation which has the effect that the `Performdef` property which is represented as a multiline string, is converted to a property called `performdef` (with lowercase name) whose value is the corresponding datastructure representation of the definition. Failure to use this assignment has the effect that the command is considered as undefined.

4 The Outcomes of Actions

One of the nice things about Leonardo is that the same action concept can be used in several ways: in the command-line dialog as has been shown above, but also in the definition of dynamic web pages and in the definition of message-passing between individuals. For example, the `newfruit` command in the example can also be invoked from a browser using the following URL:

```
http://localhost/newfruit?name=blueberry&color=blue&plant=bush
```

where the use of `localhost` is of course just an example. Moreover it is possible for one Leonardo individual to send a command for execution in

another individual using an action such as

```
[send-to-agent fruitserver [newfruit :name blueberry
         :color blue :plant bush]]
```

The present section and the next one specify what is needed in order to get these options to work as intended.

4.1 The Outcome Record

In order to make this multiple use of commands possible, it is necessary to separate the definition of how an action is performed from the definition of how its result is presented. The execution of the performance definition, such as the `Performdef` shown above, always produces an *outcome record* which in turn is handed to the appropriate forwarding or presentation method for the situation at hand. There are in principle three presentation methods: one if the command is used in command-line mode, one if the command is issued from a web browser, and one if the command has been sent as a message from individual to another and a return message is required.

The following are some major types of outcome records:

```
[ok:]
[fail: error-id "Explanation"]
[result: :type rt ... ]
```

The `ok:` record simply signifies that the action was executed successfully. It is obtained as the outcome of an action that has been defined using a `Performdef` property unless an error occurs during execution or the outcome is reset to a result record. If a subaction on any level fails, i.e. its outcome is a `fail:` record, then the superior action returns the same `fail:` record unless the error is caught and the outcome is reset. The following is an example of a `Performdef` definition where the outcome is specified explicitly:

```
@Performdef
  [soact
     [put .p has-color .c]
     [set-outcome (if [equal .c black]
                      [fail: setcolor "Cannot assign black color"]
                      [result: :value <.p .c>] )]]
```

This definition takes two arguments, `.p` and `.c`, and assigns the latter as the `has-color` attribute of the former. It returns a `fail:` outcome if the color is black and a `result:` outcome otherwise, using the verb `set-outcome`.

4.2 Presentation of the Outcome Record

The command-line executive receives an action record, executes it and obtains the resulting outcome record, and presents the results in a standardized way. In particular, if an `ok:` record is obtained then nothing is shown; the executive proceeds directly to the prompt for the next command.

If a command is sent from one individual to another, then the return message contains the outcome record in standard Leonardo notation. The presentation of, or further acting on the result is then up to the client. By exception, special rules apply when the result contains large, non-Leonardo data, for example a full text file.

Answer records headed by `ok:` and `fail:` have a fixed structure. For answer records headed by `result:` it is recommended but not always necessary to include a `:type` parameter, for example as in

```
[result: :type symval :value Italy]
```

The value of the `:type` parameter is used for specifying what other parameters may occur and what datatype they have - string, symbol, and so forth. This is presently only used if the outcome record is eventually given to the command-line executive for presentation, since then only those parameters that have been declared in the type will be displayed. Other uses of this type declaration, for example for type checking, are obviously possible but have not been implemented.

When a Leonardo server individual receives a request from a web browser then a response can only be obtained if the verb leading the request has both a *performance definition* and a *presentation definition.* The performance definition can be either a `Performdef` script like the ones shown above, or a definition in the host programming language. The presentation definition specifies what HTML contents are to be sent back for a given outcome record from the performance definition.

4.3 Definition of Parameter Types

One more thing is needed in order to get a verb definition to work in server mode, namely, a definition of parameter types. The following is how it may be written for the `newfruit` example:

```
-------------------------
-- newfruit

[: type leoverb]
[: typemap {[: name Symbol][: color Symbol][: plant Symbol]}]
```

followed by the `Performdef` definition like above. The reason why this is necessary is because the Leonardo system distinguishes between symbols, strings and numbers, but the distinction between these is lost when the parameters are transmitted in a dynamic URL.

4.4 Catching Outcome Records

In some situations one wishes that an action shall catch the outcome record of a subordinate action and proceed conditionally using its information. This requires two additional constructs, called `after` and `on`. The following is a simple example of the definition of a client-side script that sends a message to another individual, `bacchus`, and displays the result.

```
@Performdef
[after [send-to-agent bacchus [wine-advice :type suggestions
```

```
        :dish roast-beef :approx-price EUR-9]]
  [on result: [set-outcome [result: :type symval :value .answer]]]
  [on fail:   [send-to-agent lucullus ...  ]] ]
```

Here, `suggestions` is the message type specifying the datatype for the two parameters called `:dish` and `:approx-price` . It is assumed that if the remote operation is successful it will return an outcome record such as

```
[result: :answer "Rioja"]
```

containing a parameter that is then used for producing the outcome of the entire `Performdef` as

```
[result: :type symval :value "Rioja"]
```

If the `bacchus` server reported a failure, or if the communication with it failed, then a `fail:` outcome record is obtained, in which case a message is sent to the `lucullus` server instead, in the example.

In general, an expression headed by the operator `after` shall have a first "argument" that is an action, and one or more additional arguments that specify what to do for the alternative outcomes, usually through records of the form

```
[on record-former action]
```

that are used in the obvious way. Notice that the parameters of the response message are available in the action part of the `on` -expression.

5 Defining a Server Individual

This section defines how to activate a Leonardo session so that it becomes a server individual, and how to define its responses to queries. It was convenient to implement the server facility since all the basic functionality is already present in the underlying ACL system.

The server facility is not available for users in the `romulus` individual since it would be hard to administrate the access to ports. Exercises with this facility can therefore only be done in `remus` individuals on PC systems at present.

In order to start it in a `remus` individual one simply executes the following actions

```
loadk indiv-kb
awir
```

The command `awir` stands for "activate webservice in remus individual"; it starts the web server on port 80. More general commands are defined in the full system documentation. See also the separate section on the local service webpage later on in this report.

5.1 Presentation Definition

Now we wish to be able to open the web browser on the computer where the Leonardo session is running and to enter the following URL:

```
http://localhost/newfruit?name=blueberry&color=blue&plant=bush
```

This expression is quite analogous to the command-line in the above example. The desired effect is achieved by the `Performdef` definition and the parameter declarations that were shown above, except that in order for this to work correctly there also has to be a definition of the expression that is to be sent back to the web browser in response to the input just shown. This is done using an additional definition, and the full definition for the entity `newfruit` may therefore look as follows:

```
-------------------------
-- newfruit

[: type leoverb]
[: typemap {[: name Symbol][: color Symbol][: plant Symbol]}]

@Performdef
[soact [put .name type fruit]
       [put .name has-colors {.color}]
       [put .name plant-typ .plant]
       [addmember (get fruits contents) .name]
       [writefil fruits]
       [set-outcome [result: :type symval :value .name]] ]

@Displaydef
[body [heading "Thank you" :level 3]
      "Your contribution of valuable information about" .name
      "is greatly appreciated by our website." ]
```

The `Displaydef` definition is written in the Document Scripting Language (DSL) which is a variant of the Generic Scripting Language (GSL) that is used for the `Performdef` definitions.

Notice that the `Performdef` definition has been amended with two more subexpressions. First, `[writefil fruits]` whereby the newly entered fruit information is actually saved in the textual manifestation of the entityfile. In this context, notice how convenient it is to be able to use the same command library on the command line and in the definitions of scripts, as well as for the various modes of server access.

Secondly, the `set-outcome` expression defines an outcome from the performance definition that is not merely an ok record but a record containing the `name` parameter that can then be picked up and used in the `Displaydef` presentation script.

5.2 Input Forms

The previous subsection showed how the input to a dynamic webpage was given directly as a URL containing parameters. In practice one wishes of course to provide such parameters using a form. The following is an extension of the example whereby the user may visit

```
http://localhost/newfruit-specify
```

and obtain a form that can be completed and sent to the dynamic webpage that was shown above.

```
-------------------------
-- newfruit

[: type leoverb]
[: typemap {[: name Symbol][: color Symbol][: plant Symbol]}]

@Specifydef
[body [heading "Entry of Fruit Information" :level 3]
      [request :to newfruit :method post ^
          [table :frame box :rules all :border 1 ^
              [row [box "Name:"]
                   [box :charsize 50 ^
                        [input text :tag name] ]]
              [row [box "Color:"]
                   [box :charsize 50 ^
                        [input text :tag color] ]]
              [row [box "Plant-type:"]
                   [box :charsize 50 ^
                        [input text :tag plant] ]] ]
          [sendbutton "Enter"]
            ]]

@Performdef
[soact [put .name type fruit]
       [put .name has-colors {.color}]
       [put .name plant-typ .plant]
       [addmember (get fruits contents) .name]
       [writefil fruits]
       [set-outcome [result: :type symval :value .name]] ]

@Displaydef
[body [heading "Thank you" :level 3]
      "Your contribution of valuable information about" .name
      "is greatly appreciated by our website." ]
```

The `Specifydef` definition uses the Document Scripting Language (DSL) like the `Displaydef` definition. Notice that it is vaguely similar to HTML, although some of the non-mnemonic tags in HTML have been replaced by more readable ones, such as `row` and `box` for the substructures of a table. The DSL notation is however much more powerful due to the possibility of introducing control structures such as loops and conditionals, and through the use of DSL scripts which serve as macros, and finally through the use of composite expressions such as sequences and sets.

The example just shown is an instance of a frequently arising situation where there is one webpage containing a form for the user to complete, a set of operations that are to be done for the given input, and a new webpage that is to be produced as a result of the input and the operations. All three steps can be organized in one and the same entity, but the first one of the two webpage descriptions is given a separate name. In the example, where the entity in question is `newfruit`, the first webpage definition is attached to the identifier `newfruit-specify` and the second one as well as the `Performdef` is attached to the identifier `newfruit`. This is why the page containing the form is accessed as `http://localhost/newfruit-specify`, and it is also why the definition of the form specifies `newfruit` for the `to` parameter of

its `request` subexpression.

The system is however by no means restricted to such threetuples of definitions. A `request` expression may refer to arbitrary other webpage entities in its `to` parameter, which gives full flexibility.

5.3 State Transformations

Although the `Performdef` script of a dynamic webpage will often make changes in the current information state of the server, for example using the `put` command, there are also frequent situations where this script should calculate a value that is then sent to the presentation script. The value may be obtained from input data elements, or from the contents of the current information state of the server, or from a combination of those.

Suppose for example that the current information state of the individual already contains information about the major exporting countries of various fruits, and that one wishes to present this information to the user. One way of doing this would be as follows, where the earlier definition is amended and the `Specifydef` script is omitted:

```
-------------------------
-- newfruit

[: type leoverb]
[: typemap {[: name Symbol][color Symbol][plant Symbol]}]

@Performdef
[soact [put .name type fruit]
       [put .name has-colors {.color}]
       [put .name plant-typ .plant]
       [addmember (get fruits contents) .name]
       [writefil fruits]
       [set-outcome [result: :type fruit-report :value .name
                      :in-countries (get .name exported-from)] ]]

@Displaydef
[body [heading "Thank you" :level 3]
      "Your contribution of valuable information about" .name
      "is greatly appreciated by our website."
      "The major exporting countries of this fruit are"
      [enumerate-list :list .in-countries] ]
```

Here the outcome record from the performance definition is used not merely for forwarding an input value to the presentation script, but also for adding further information from the database. It would of course have been possible to use the expression `(get .name exported-from)` directly in the presentation script, but keeping all data access in the performance script may be useful, particularly if the same verb is used in several access modes and not merely from a browser.

In the particular case of message-passing between individuals the presentation script is irrelevant since the outcome record is returned as is from the server to the client. In such cases one may think of the action verb as a state transformation, with the parameters of the primary message as the

incoming state and the parameters of the outcome record as the outgoing state.

The examples assumes that the value of e.g. `(get blueberry exported-from)` is a set of entities and that the verb `enumerate-list` takes a set for its `list` parameter and produces that list with appropriate formatting and interpunction.

The example also makes use of a specialized type for the outcome record, namely, `fruit-report`. In general, every outcome record type must be declared like in the following example

```
-------------------------
-- fruit-report

[: type tftype]
[: typemap {[: value Symbol][: in-countries Set]}]
```

The available type designators include `String`, `Symbol`, `Integer`, `Set` and `Sequence`. This is an intermediary solution; it is intended later on to use the type-specification machinery that already exists for other purposes in the full Leonardo system but not so far in the simplified Leordo system.

In more complicated cases one may wish to assign intermediate data values to a local "variable" for use later on in the performance script. This can be done using the operator `set` which effectively considers the set of incoming parameter assignments in the performance definition as a local state that it is able to modify. The `set` operator takes two arguments where the first one is the state component, or parameter, that is to be assigned or reassigned, and the second argument is its (new) value. It can therefore be used both for changing the value of an already existing parameter, or state component, and for introducing an additional one.

The first argument of `set` is specified without an introductory colon or point. For example, the subaction

```
[set in-countries (get .name exported-from)]
```

in the above example would introduce an additional parameter `:in-countries` and assign a value to it.

Notice, however, that all parameters and assignments are local to the performance definition, and if some of them are to be made available to the presentation definition then this has to be done using the outcome record.

6 Web-based Support for Remus Individuals

6.1 The Local Webpage for User Service

One of the uses of the webserver support in Leordo is for a simple webpage that runs in `localhost` mode and provides the session user with miscellaneous services. The development of this facility has barely begun, but we mention it here as an illustration of what is possible. It is intended to provide both information about the current system, access to applications that have been developed in the individual at hand, and also personal services for the user.

The user-service webpage is activated by the `awir` command that was defined above, and it operates on port 38081. In a session for a Remus system (on your local PC) where this command has been issued, just open the web browser and visit

```
http://localhost:38081/
```

This will open a page that is maintained by the session at hand, and that can provide access to various services that are defined in it.

Notice however that the `awir` command activates both port 80 and port 38081. Port 80 is intended for services that the user wishes to develop and use externally; port 38081 is for the local service webpage. If only the latter is desired then the command `alwir` should be used instead.

If you should wish to deactivate the web server, issue the command

```
serstop
```

The command `awir` or `alwir` will reactivate the server.

6.2 Registration of Remus Clones

Each Leonardo individual has both a *name* and an *identifier*. The identifiers are supposed to be unique for each individual; the names are chosen by the user at the time when the individual is bred, and do not have to be unique. For the details of this scheme, please refer to the general Leonardo documentation.

The Romulus and Remus individuals have these respective names in the Leonardo sense, and different identifiers as well. Distributed copies of the Remus individual have the same name and also the same identifier, because of the way they are produced. If and when one starts using such cloned copies for message-passing or other interaction between individuals it is desirable to change the identifier so that the clone in question obtains its own, unique identifier. This is done using a web-based registration service. Please refer to the general Leordo information page [4] for details.

6.3 Obtaining Additional Knowledgeblocks for Remus

The Leordo information page also contains additional knowledgeblocks for download. Each new knowledgeblock is obtained as a separate directory which is to be placed on the top level of the individual, beside the initially included knowledgeblock directories. In addition, to incorporate the downloaded knowledgeblock `my-new-kb` in an individual, one has to execute the command

```
defk my-new-kb
```

in a session with it. This command has the effect of amending the individual's register of available knowledgeblocks with the additional entity. It differs from the `crek` command which was defined above since `crek` also initializes the contents of the knowledgeblock.

[4] http://www.ida.liu.se/ext/leordo/

7 Message-Passing Details

This section explains how messagepassing is done, using the http protocol and the webserver that was described in the previous section.

7.1 Message Formats

The message traffic is done in server/client mode, so there is always a *primary message* from individual A to individual B, and a subsequent *response message* from B to A unless, of course, some problem prevents this from happening. One and the same individual can serve alternatingly as server and as client. Primary messages are expressed as `http` queries in the same way as was shown in an earlier section, but with an additional parameter called `i2i` (for "individual to individual") in order to indicate that an outcome record is requested as a response. Outcome messages are expressed as short text messages like in the following example:

```
<html><body><pre>
!-----------------------------------
[result:
    :name pineapple
    :in-countries {Thailand Philippines Indonesia}]
!-----------------------------------
</pre></body></html>
```

This is for a hypothetical scenario where the primary message requested information about what are the world's largest exporters of pineapple. Both the primary message and the response message is therefore essentially a record (an action record and an outcome record, respectively), although with different concrete representations. The use of an HTTP wrapping around response messages has been chosen in order to facilitate debugging. The implementation has of course been done in such a way that the representation of the messages can easily be changed.

7.2 Use of the Host Programming Language

Besides defining performance and presentation operations through scripts, there are also handles in the Leonardo system for defining these operations using program stubs in the host programming language, which is CommonLisp at present. The separate system documentation describes how Leonardo verbs can be defined on the Lisp level.

Actual applications to date have relied to a large extent on Lisp-level definitions for performance definitions. Our strategy in this respect has been to first implement applications in the host language in order to gain experience with what facilities may be needed, and then to migrate gradually to using the higher-level script languages.

For presentation definitions, on the other hand, the Document Scripting Language has been used consistently. Notice, by the way, that we also use a variant of DSL as the source notation in the preparation of reports and articles, such as the present one.

8 Conclusions

This report has described the first steps towards using Leordo which is a small and simple Leonardo system. The Leordo website [5] contains links to reports that describe additional uses of Leordo.

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