Leonardo

Leonardo Project
Division for Artificial Intelligence and Integrated Computer Systems (AIICS)
Department of Computer and Information Science, Linköping University

Erik Sandewall

Leonardo Installation and Usage

This series contains technical reports from the Leonardo project, for the development of a software infrastructure for knowledge-based systems.

The present report, PM-leonardo-017, can persistently be accessed as follows:

Project Memo URL:	http://http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/017/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2012/001/
Date of manuscript:	2012-09-14

Related information can also be obtained through the following www sites:

Leonardowebsite:	http://http://www.ida.liu.se/ext/leonardo/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/∼erisa/

Chapter 1

Introduction to the Leonardo System

The present report is a practical introduction to the Leonardo software system – its intended applications and mode of use, the basic approach to its design, how to acquire a copy of the system and how to install it, and also how to write simple scripts that define its behaviors. For each of these aspects of the system there exist additional documentation that go further into the details.

Up-to-date information about the Leonardo system is also available using its websites, which are

```
http://www.ida.liu.se/ext/leonardo/
http://www.ida.liu.se/ext/leordo/
```

for the Leonardo project and the actual system, respectively.

1.1 Characteristic Properties

The Leonardo system is a software platform for *knowledgebased systems,* that is, systems that make active use of structured information about some aspect of the world. An important area of use is as a platform for *cognitive systems,* that is, systems where structured information about the system's environment and about itself is used for guiding its choice of *actions* in that environment. Our approach to cognitive systems is described in our recent textbook, *Artificial Intelligence and the Design of Cognitive Systems.* [1]

Cognitive systems make use of a number of techniques from Artificial Intelligence and Knowledge Representation, including particular algorithms and designs for software "engines." The Leonardo software platform does not in itself contain these artifacts; it is instead intended as a suitable basis for implementing and using them. However it is also used for, and intended to be used for a variety of other kinds of knowledgebased systems which do not necessarily make use of A.I. techniques.

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

1.1.1 Standard and Salient Characteristics

The characteristics of the Leonardo system include some standard properties that are unsurprising and can be found in many places, and some salient properties that are relatively unusual and that may be the effective reasons for deciding to use this particular platform for an application. The following are the most important standard properties.

- The Leonardo platform provides an environment for incremental development and execution of programs and scripts, in the tradition of programming languages such as Lisp and Python.
- It provides and makes use of a knowledge representation language (CEL) that subsumes first-order logic and elementary set theory notation, but which also contains a number of extensions to these.
- Interactive sessions with an instance of the Leonardo system are performed using a combination of a command-line executive and a graphic user interface (GUI) which operate concurrently, much like a user can give commands to an operating system alternatingly using a "terminal window" and a GUI.
- One of the uses of the CEL is as a scripting language for defining command-line macros and for defining additional interaction "pages" in the GUI.

The most important salient property of a Leonardo system instance is that it is *time-aware, action-aware,* and *communicable.* Its knowledge representation system includes explicit representation of calendar time and of actions, including both its own actions, the actions of other Leonardo instances, and the actions of its user(s). This is used in the following ways.

- Instances of the Leonardo system can exchange messages between each other, in particular using the CEL language for knowledge representation. A message may be eg. a request for information, or a request to perform a particular action.
- Each instance of the system contains a model of its computational environment, including the properties of, and relations between computer hosts, local area networks, memory devices, information servers, and so forth, as well as properties of other Leonardo instances. This means that the infrastructure for message exchange is represented explicitly and available for deliberation.
- Each instance of the system also contains a representation of its own past history, including information about its own origin, and information about actions that the system has performed in the past.
- A script that defines the execution of an action in terms of simpler actions, or using the system's host programming language, consists in principle of three parts: a specification of the action's preconditions, a specification of how to execute the action itself, and a separate specification of how to present the outcome of the action to a user. This partitioning of the definition is essential eg. when a Leonardo instance decides to delegate the execution of an action or sub-action to another instance of the system.

- The system's command-line executive uses the three parts of an action definition in succession, but if the precondition is not satisfied then the system has a variety of ways of identifying repair actions that achieve the precondition, or alternative actions that may be used instead.

One consequence of these properties is that Leonardo systems are well suited as a platform for a set of *delegating cognitive agents,* ie. cognitive systems that can delegate tasks to others and receive delegation of tasks, and do this in an orderly manner.

Another salient property of a Leonardo system is that it is designed for *very long time use,* where an instance of the system may evolve for months or years. This property is important in software that learns in serious way; this will be discussed next.

1.1.2 The Software Individuals Paradigm

The Leonardo system uses the *software individuals* paradigm. A software individual is a software artifact that is capable of evolving over time, using for example knowledge acquisition or machine learning, and that has an identity so that it is well defined what is "the same individual as" or "a different individual from" a given one, even if the individual is moved from one location to another.

Informally one should therefore think of a software individual as a cognitive agent, ie. one that is able to plan its actions, work towards given or chosen goals, communicate with other software individuals using message-passing, communicate with people using web pages and its built-in webserver, or by sending email to them, retain a history of its own past actions and use that history for its continued activity, and so forth. Although the Leonardo system does not (yet) contain implementations of the Artificial Intelligence techniques that will be needed for this, it does provide many of the required practical facilities, such as agent-to-agent communication, communication with users by email, management of passwords, operating a web server, and so forth.

Each individual has a *persistent manifestation* consisting of a number of directories and files in a computer memory. When a computational session is started based on this information then that session is considered as a *dynamic manifestation* of the same individual. The persistent manifestation is organized as one directory with subdirectories in the present implementation, although in principle one could also have chosen other ways of organizing its collection of data. Sessions with the individual will regularly change the contents of the persistent manifestation, and this is how the individual can evolve over long periods of time.

The concept of *identity of individuals* is defined in terms of the persistent manifestation. If one makes a full copy of the persistent manifestation in the filesystem where it is located, then one obtains a *clone* which can be personalized and become another individual, but in any case it is distinct from the individual that was copied. On the other hand, if one makes a move of the persistent manifestation, for example from a hard disk to a USB stick memory, then one still has the *same* individual although in the new location.

The notions of persistence and identity of individuals are important from the point of view of Artificial Intelligence. Machine learning is only of interest if the entity that learns has a sufficiently long "life" so that it can accumulate knowledge gradually and have a chance to use it. Also, communication between agents and delegation of tasks is only possible if each individual has a name and a location where it can be reached, and if it is clear which computational process is to perform the actions that a particular agent has committed to.

Persistence and identity of individuals is also related to the issue of *mobility.* Leonardo individuals are "mobile agents" in the sense that the persistent manifestation can be moved from one computer to another, so that the dynamic manifestation can take place in different computer hosts at different times, retaining the concept that it is *the same individual* that just moves from one host to another.

A Leonardo-based software individual will often be referred to as an *exemplar* of the Leonardo system, rather than eg. an "instance" of it, in order to emphasize that different individuals may develop in different ways, so that individuals that were created as clones and that had identical contents at the outset may be quite different when they have evolved for some time. [2]

The way to start using a Leonardo system is therefore to obtain a copy of a simple Leonardo individual and to gradually fill it with contents, first through commands that are given to it by its owner, and later on possibly by more or less autonomous activities in the individual. Some of the initial commands should be commands that import and install software modules for particular *software facilities* that may be obtained from the Leonardo library, or fact and knowledge modules that may be obtained from the Common Knowledge Library [3]

1.2 Personalization and Registration

A Leonardo individual can describe itself as being in either of three states: *individualized, personalized,* or *registered.* Through interaction with its user and with other individuals it may change the state from individualized to personalized, and from personalized to registered.

If a copy is made of the the individual's persistent representation (usually by making a copy of a directory in the computer's file system, with all its subdirectories and files) then the copy retains the choice of state, but it also changes its status from *original* to *copy.* Therefore the persistent manifestation may be an individualized original, an individualized copy, and so forth.

If a computational session is started in an individual with copy status, then the session startup process will recognized the situation and change the state and status almost automatically to an individualized original. This change is called *individualization* and it has the following effects in the individual:

[2] In the English language the word 'exemplar' has a dual meaning, so it can refer both to one out of many copies of eg. a book, and also to a prototype that copies are made from. We use the word in the former sense.

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

- Remove data that pertain to the history of the individual that was copied.
- Requires the user to enter a *master password* for the new individual. This password will be required for performing certain operations in the individual.

The operation of *personalization* is invoked by the user. It changes the state from individualized to personalized, and has the following effects:

- Introduce entities and attribute values whereby the individual characterizes its owner and its physical location.

The operation of *registration* is also invoked by the user, and requires that the individual is able to communicate with a particular *registrar* individual, which usually means that it must be connected to the Internet. It changes the state of the individual from personalized to registered, and has the following effects:

- The individual informs the registrar about its existence.
- The registrar assigns a name to the new individual, for use within the namespace that the registrar administrates.
- The individual receives up-to-date information from the registrar concerning the current state of the computational environment. The individual needs this information for the purpose of message-sending, including querying other individuals and delegating actions to them.

An individual that has been obtained as a copy of another individual and subjected to at least the first one of these operations, will be referred to as a *clone* of the individual that it was copied from.

Besides the operations mentioned here, there is also an operation called *accomodation* that occurs when a Leonardo individual starts a session in a host where no Leonardo individual has executed before. This operation sets up a description of the host at a designated location in the host itself. Accomodation will be described in Chapter 3.

1.3 Individuals, Agents, and their Names

The Leonardo platform can be used for a variety of applications, and we therefore distinguish between different *species* of individuals. Each species is characterized by a distinct internal organization within the framework of the general Leonardo architecture. In some species there is an *original exemplar* that is the primary one for the purpose of software development, so that other exemplars of the same species can be updated by copying information from the original exemplar. Other species may allow different individuals to evolve independently in different and diverging directions.

Each individual has a unique *name* or, more specifically, a name that is unique within a given namespace. (More about namespaces later). These names are formed as consisting of a species name and a serial number, for example `Registrar-2` for one individual in the `Registrar` species. The name is assigned by a registrar when an individual is registered.

Each individual consists of a number of *agents.* The partitioning into agents serves several purposes, including software modularity. Each agent belongs to a *variety,* just like individuals belong to species. For example, an individual of the `Pawn` species contains one agent of each of the `remus, demus,` and `utilus` varieties. `Pawn` individuals have an elementary character and they are used for initial exercises by new users, as well as for testing replication and other basic system facilities.

The agents, in the sense of the Leonardo architecture, are therefore building-blocks that individuals are constructed from. Each agent variety may be used in more than one species. In particular, each individual must have a *kernel agent* containing the basic software for organizing an individual. At present there is only one variety of kernel agents, called `remus,` so every individual must contain an agent of the `remus` variety.

Besides consisting of a set of agents, an individual must also contain a *model of its own basic properties* as well as a *map of its computational environment.* The latter is a structure that describes what other individuals there are, what computer hosts or detachable memory devices they are located on, what local area networks are in use and what computers are located in each of these, and so forth. This information is needed in order to support communication between individuals so that, for example, an agent in one individual can perform the action of sending a message to an agent in another individual with a specific name, and the system will keep track of where that individual is located and how it can be reached. Each individual contains its own copy of this map, and it is updated when the individual registers and during regular interactions with the registrar. This map is referred to concisely as the individual's `indivmap` since its primary purpose is to provide information about related individuals.

1.4 Starting a Session with a Leonardo Individual

Much of the above will become more clear by experimenting with a simple Leonardo individual. The reader should obtain a copy of a `Pawn` individual; the following is what he or she will find.

The individual arrives as a directory with subdirectories, where the top-level directory has a name eg. `Pawn-17.` It has five subdirectories, called

```
Indivmap
Indiv-self
remus
utilus
demus
```

The `Indiv-self` directory contains the individual's model of itself, and the `Indivmap` directory contains its model of its computational environment. Changes in the self-model are first made in the contents of the `Indiv-self` directory. This can happen both off-line and on-line. When the individual is connected to its Registrar it can upload the changes in the self-model to the Registrar, who then distributes the updated model of the environment (ie., contents for the `Indivmap` directory) to other individuals as well.

The remaining three directories represent the three agents in the individual. The `remus` agent is the kernel, `utilus` is a library of utilities, and `demus` is used for test and demo purposes.

It is recommended to introduce one particular directory, normally called `Leonardo,` that is dedicated to containing one or more such individuals. The examples in the sequel witll refer to directory levels relative to such a surrounding `Leonardo` directory.

Looking inside the agents, each of them has a subdirectory called `Process` and a path such as

```
Leonardo/Pawn-17/demus/Process/main/
```

leading to the *invocation directory* for the agent in question. This is the directory level where computations with this agent will start. It contains a few *invocation files* that are used for starting sessions in different environments, such as the file

```
winleo.bat
```

for starting a session in a Windows environment, and

```
acleo.lex
```

for starting a session in a Linux environment and using Allegro CommonLisp. A session in Linux using CLisp is started using `cleo.lex.` Thus in Linux the session may be started from the invocation directory by doing

```
./acleo.lex
```

The predefined contents of these invocation files may need to be modified in order to reflect the actual location of the Lisp system being used. Please refer to Chapter 2 for information about which implementations of Lisp are supported by Leonardo, and how to obtain one if it is not already installed in the computer at hand.

When a session is started in either of these ways, one obtains a command-line window with the following kind of text (one newline introduced here for linelength reasons)

```
;   Loading C:\Leonardo\Pawn-74\demus\Startup\cl\bootfuns.leos

****************************************************************
Now starting up a Leonardo session  --  please stand by

Welcome to a session with the Leonardo software individual Pawn-74
The session is done with the individual's agent called demus
This is session number 29 with this agent

This session executes on the following Lisp system:
International Allegro CL Enterprise Edition
8.1 [Windows] (Jul 28, 2007 7:42)
Copyright (C) 1985-2007, Franz Inc., Oakland, CA, USA.
   All Rights Reserved.

This development copy of Allegro CL is licensed to:
   [pc0756] Linkoping University
```

```
****************************************************************
ses.001)
```

The initial text under CLisp is different, but not in any essential way. Each agent keeps track of how many sessions it has started; this counter can be seen in the file

```
Leonardo/Pawn-17/demus/Process/main/State/session-nr.txt
```

The session number is used for maintaining the individual's history of its own activities. It is reset to 0 when a copy is individualized and becomes a separate individual.

After the startup message the session may proceed to a command-line dialog where each interaction uses a prompt of the form

```
ses.001)
```

where `ses` indicates that the choice of "listener" for commands is the main session listener, and the following number shows the interaction number for that listener. In particular situations the agent may switch to using other such listeners.

If the system is run under Allegro CommonLisp then the command-line dialog starts automatically at the end of the session startup routine. If it is run under the CLisp implementation then the user must invoke the command-line executive by typing

```
(cle)
```

to the Lisp system's read-eval-print loop.

The other agents, besides the `demus` agent, contain the same invocation files in the corresponding locations, with minor differences. The `demus` agent is defined in such a way that it is suitable for initial experiments and demos.

1.5 Simple Command-line Input in Leonardo

The remainder of this chapter will describe the character of command-line input to a Leonardo individual and the organization of information in it. This is relevant by way of introduction since it captures essential aspects of the system's character, but also because Leonardo individuals are consistently self-describing, which means that the installation and configuration procedures for Leonardo make use of the system's general-purpose information structures. The Session GUI makes use of the same information structures and will be described in Chapter 3.

There are four kinds of inputs to the Leonardo command-line loop: commands, CEL forms, Lisp forms, and CEL propositions. Both kinds of forms are expressions that evaluated in order to obtain their value. The following are examples for CEL forms:

```
ses.024) (+ 4 5)
  => 9

ses.025) (str.concat "abc" "def")
  => "abcdef"
```

where `str.concat` is the string concatenation operator in CEL, the Common Expression Language. The separate CEL manual defines the reportoire of functions and other operators in CEL.

Since Lisp is the host language of the present Leonardo implementation it is possible to also input Lisp forms and have them evaluated directly; they shall be preceded by a point (full stop) on the command line in order to distinguish them from CEL form input, as in the following examples.

```
ses.026) . (+ 4 5)
9

ses.027) . (cdr '(red blue green))
(blue green)
```

Some usage of Leonardo can be made using the CEL language, but in many cases it is necessary to also implement operations in Lisp, and direct execution of Lisp forms is often needed for testing and debugging in such cases.

Commands may be simple or composite. A simple command in command-line input is written as a command verb followed by its argument(s) and parameter(s). For example, the `put` command with three arguments assigns the third argument as the attribute value for the combination of the first two arguments, for example:

```
ses.028) put gunnar has-children {karin bertil inger}
put: gunnar has-children {karin bertil inger}

ses.029) (get gunnar has-children)
  => {karin bertil inger}
```

The CEL `get` function obtains the attribute value for the combination of its two arguments. The elementary `put` command is defined so that it always prints out a message confirming that the attribute assignment has been done. This is done even when the `put` command is part of a larger, composite command, as in the following example:

```
ses.034) soact [put a color red][put a size 41]
put: a color red
put: a size 41
```

The operator `soact` stands for "sequence of actions" and it is used to combine a number of commands that are executed sequentially. In CEL notation, every command per se is surrounded by square brackets, as in the following examples:

```
[put gunnar has-children {karin bertil inger}]
[soact [put a color red][put a size 41]]
```

and it is simply a convention of convenience in the command-line loop that the outermost level of square brackets shall be omitted for commands.

Propositions may also be simple or composite. A simple proposition, also called a literal, is formed as a predicate followed by its arguments, and enclosed in square brackets even in command-line input, as in the following example:

```
ses.030) [member bertil (get gunnar has-children)]
  => [true]
```

The value shall be `true` if this expression is evaluated following the `put` command above. Composite propositions are formed using propositional operators such as `and` and `or`

```
ses.031) (or [member bertil (get gunnar has-children)] [member mia {}])
  => [true]
```

Consider also the following example:

```
ses.032) (and [member bertil (get gunnar has-children)] [member mia {}])
  => [false]
Culprit literal: [member mia {}]
```

This shows how the evaluation of a conjunction is defined in such a way that if the conjunction is false, the system identifies the conjunct (or one of the conjuncts) that falsified the conjunction. This feature is important eg. when action preconditions are evaluated, since it can be used for selecting preparatory actions that achieve the missing part of the precondition.

Notice that both commands and literals are surrounded by square brackets, and both terms and composite propositions are surrounded by round parentheses. Type declarations for the operators leading the respective expressions determine the category of each expression.

The propositional operator `not` may be used in the same way as `and` and `or` but for the negation of literals one may also use the convention that is shown in the following example

```
ses.036) [-member bertil (get gunnar has-children)]
  => [false]
```

where in general, if a single dash character is prepended to a predicate symbol then the negated predicate is obtained.

The session is terminated in a regular manner by entering the following command

```
exit
```

However, just terminating the session job does not have any detrimental side-effects except possibly in some specialized applications.

If it should occur during a session that the system runs into a Lisp-level or lower-level error that is not caught by the Leonardo software, then the command-line dialog reverts to the Lisp level. In such cases one can return to the Leonardo command-level dialog by entering

```
(cle)
```

where `cle` stands for "command-line executive."

1.6 Knowledgeblocks and Entityfiles

The capability of storing information for later use is essential for a software individual. The `put` command shown above is not persistent: if it is performed within a session then the assignment is retained during the continued session or until the attribute value is reassigned, but it is lost when the session ends.

The primary method for preserving information in the persistent manifestation of an individual is by using *entityfiles.* An entityfile consists of a sequence of *entity descriptions,* where each entity description consists of an entity and one or more assignments of attribute values for this entity. Entities may be symbols or composite entities.

We use the term 'entityfile' both in the abstract sense of the sequence of entity descriptions, and in the concrete sense of a text file in the directory structure of the individual's persistent manifestation. The following is a fragment of an entityfile in the latter sense:

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

[: type thingtype]
[: attributes <has-color plant-type produced-in>]
-----------------------------------------------------------
-- pineapple

[: type fruit]
[: has-color orange]
[: plant-type bush]
[: produced-in {thailand nicaragua}]
-----------------------------------------------------------
```

The example shows how the entity description for the entity `pineapple` indicates its `type` and the values of its three attributes, and also how the type itself ie. `fruit` is also defined as an entity, which itself has a type and other attributes. The `attributes` attribute of `fruit` specifies what attributes may be defined for an entity whose type is `fruit.`

The syntax for entityfiles is defined with a tradeoff between readability and simplicity. In particular, the use of a line of dashes for separating successive entity descriptions was chosen rather than a conventional (for computer science) structure of *begin* and *end* tags simply because it makes it easier to quickly obtain an overview of the contents of a textual entityfile.

A few additional syntactic conventions are used in entityfiles, in particular for attributes where the value is a long string, ie. a string consisting of several lines of text. These attributes are used both for source code in the host programming language (ie. Lisp), and for documentation and other commentary. However this need not concern us here.

Entityfiles are further organized into *knowledgeblocks* each consisting of a *master file* for the knowledgeblock and a number of additional entityfiles. The following commands are used for working with these, with examples of arguments from the fruit domain used above. The abbreviation ef stands for entityfile and kb for knowledgeblock.

```
crek fruits-kb        Create a kb for fruits information
setk fruits-kb        Select this as the current kb
crefil fruitsdef      Create an ef in the current kb
loadfil fruitsdef     Load the ef called fruitsdef
writefil fruitsdef    Write back the ef called fruitsdef
```

For example, if the ef `fruitsdef` contains the entity descriptions shown above, and one executes the following commands

```
loadfil fruitsdef
put pineapple has-color brown
writefil fruitsdef
```

then the result will be that the entitydescription for `pineapple` in the updated file looks as follows.

```
-------------------------------------------------------
-- pineapple

[: type fruit]
[: has-color brown]
[: plant-type bush]
[: produced-in {thailand nicaragua}]
-------------------------------------------------------
```

There is therefore an interdependence between the assignments of attribute values within the computational session and the assignments that are stated in the textual entityfiles. The information contents of an individual can evolve in two complementary ways. One way is for the user to text-edit the contents of entityfiles and then load the revised file using the `loadfil` command, the other way is for the user, or a computational process, to revise attribute-value assignments within the computational session, and then save the updated structure using the `writefil` command.

By convention, the name of a knowledgeblock must end with `-kb` and it will be located in a directory whose name is the name of the knowledgeblock without that postfix, and with a capital first letter. Thus the entityfiles `fruits-kb` and `fruitsdef` will be located in the directory

```
Leonardo/Pawn-17/demus/Fruits/
```

in the example. Knowledgeblocks and their contents may be inherited between agents in the same individual, and in particular the contents of the `remus` kernel agent are available in all the other agents in the same individual.

For each entityfile the system must know what are the entities in it. This is accomplished using the `contents` attribute for an entity representing the entityfile. In the example above, one may obtain the list of contents as follows.

```
ses.048) (get fruitdefs contents)
  => <fruitdefs fruit pineapple>
```

In general, each entityfile is represented by an entity, and the value of the `contents` attribute of that entity is the sequence of the entities in the entityfile, including the entityfile symbol itself as the first element in the sequence. The following is its entity description, with some attributes omitted.

```
-------------------------------------------------------
-- fruitdefs

[: type entityfile]
[: latest-written "2011-12-09/05:03.+01"]
[: contents <fruitdefs fruit pineapple>]
-------------------------------------------------------
```

1.7 Parsed Entityfiles

Section 1.4 mentioned the existence of files with the extension `.leos` whereas the standard extension for entityfiles is `.leo` – a few words of explanation for this is in place.

The background is that whenever an entityfile is loaded into the Leonardo system it has to be parsed, ie. its contents are converted from the CLE notation to the corresponding representation as S-expressions in Lisp, which is the representation that the Leonardo software actually uses internally. For each entityfile in CLE notation with the extension `.leo` the system can also produce a textfile with the same information in S-expression notation, ie. the textual form of the parse of the original file. These files obtain the extension `.leos` where the `s` stands for S-expression.

Moreover, since all entityfiles in a knowledgeblock are usually collected in the same directory, the corresponding `.leos` files are kept in a subdirectory of the same directory, where the name of the subdirectory is `cl`. This is done in order to avoid cluttering the directory of the `.leo` files.

The `.leos` representation must necessarily be used for the first few files that are loaded during session startup, such as for the very first file that is loaded in a session, viz.

```
Leonardo\Pawn-74\demus\Startup\cl\bootfuns.leos
```

which was used in an example above. It also applies of course for the files that contain the CLE parser itself. However, parsed entityfiles are also useful for performance reasons in the case of very large entityfiles, containing several thousand entitydescriptions.

These `.leos` files are generated at the same time as a conventional `.leo` file is written by the system, but conditionally on whether the descriptive information for the entityfile indicates that it shall be generated. For ordinary entityfiles with moderate size it is usually not meaningful to produce and use the parsed entityfiles.

This chapter has described the most important aspects of how persistent memory and command-line dialog is organized in a Leonardo individual. There are many more aspects to this, of course, and these will be described in the following chapters as well as in the other Leonardo documentation reports.

Chapter 2

Obtaining a Lisp System for Running Leonardo

Leonardo is implemented in CommonLisp, so a CommonLisp implementation is necessary in order to run it. Several such implementations exist and can be used. The present chapter describes how to choose, obtain, and start using such an implementation.

2.1 Initial Considerations

The following are some general considerations that one should be aware of, regardless of the choice of operating system and of Lisp implementation.

2.1.1 Choice of CommonLisp System as Platform

Several implementations of CommonLisp exist. The actual implementation of Leonardo was done using Allegro CommonLisp, and the system has also been modified so that it can operate in the CLisp implementation, with only minor restrictions due to the lack of certain facilities in the latter. Allegro CommonLisp is a commercial software product that is available for Windows, Mac, and Linux operating systems. CLisp is in principle also available for all of these, but for users that prefer to be in an environment where Windows-based services are available, we recommend to install a virtual environment eg. using VMware, to install Linux inside it, and then to put Leonardo there.

2.1.2 ANSI Base vs. Modern Base Lisp

When installing a Lisp system and a corresponding Leonardo agent, one should be aware of the distinction between ANSI Base and Modern Base Lisp systems. ANSI Base systems 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.

ANSI Base systems are therefore case-insensitive, to a large extent, and such was the historical practice in Lisp and it became entrenched through the ANSI Standard. ACL and CLisp offer both possibilities, but some other implementations of Lisp offer only ANSI Base.

Leonardo has been implemented in a context of Modern Base. A conversion to ANSI Base has been done but is not operational.

2.2 Allegro CommonLisp

Allegro CommonLisp is one of the two currently supported platforms for Leonardo. It is a commercial product produced by Franz, Inc. in Oakland, California. The Franz website is located at

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

The following alternatives exist for obtaining an Allegro CommonLisp system:

- Purchase a full ACL system (fairly expensive).
- Purchase a student system (if you are a student, moderately priced).
- Obtain the free 'Allegro Express' system from Franz (requires you to re-register from time to time while using the system).

The following are the details for each of these alternatives.

2.2.1 Running Standard ACL on a Desktop or Laptop

Download the system from the website shown above and follow the instructions as you proceed. Also, obtain a license file which is a small ASCII file called `devel.lic` and add it to the top level directory of the installed ACL system. It is obtained with the purchase, or from your system administrator if your institution has a site license.

The executable that is to be used for the purposes of Leonardo is found in the top-level directory and is called `mlisp` with the appropriate extension, for example `mlisp.exe` for Windows.

2.2.2 Using the Allegro Student System

From the point of view of usage as a platform for Leonardo, the Allegro Student System is used in the same way as the standard (= professional) system.

2.2.3 Using the Allegro Express System

On the Franz website, 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, Mac).

Before the Allegro Express system can be used for Leonardo it must be configured for this purpose. The system that is obtained from the download has two undesirable features: 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.

On a Windows system: Position yourself where the system is located, which is usually at

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

(for version 8.1, otherwise by analogy). Click the file `allegro-express.exe` in this directory so as to start the run. This will open a group of two windows. Close 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.

The procedure for installation under Linux or Mac is analoguous.

2.3 Adjusting Leonardo Invocation Files

Chapter 1 mentioned the *invocation directory* in a Leonardo individual, such as

```
Leonardo/Pawn-17/demus/Process/main/
```

in the example there. This invocation directory shall contain *invocation files,* ie. executable files that are used for starting sessions with the Leonardo individual in question. One and the same invocation directory may contain invocation files for running in both Linux and Windows, and for using one or the other Lisp implementations as platform, or for running on one or another computer host. A separate invocation file is used for every such combination. It is also possible to define additional invocation files for starting a session in a particular way.

A standard distribution of Leonardo contains the following invocation files for a start.

```
winleo.bat              For Windows and ACL
acleo.lex               For Linux and ACL
cleo.lex                For Linux and CLisp
```

Each of these contains a single line, such as

```
start C:\Progra~1\acl81\mlisp.exe -L ../../Startup/cl/winleo.leos
```

for `winleo.bat`. (The use of files with the `.leos` extension was discussed in the last section of Chapter 1). In every such case there is an *invocation entityfile,* such as `winleo.leo` that contains the definitions for this particular way of starting the Leonardo session. A particular generation command will take the name of such an invocation entityfile as its argument and generate the corresponding `.bat` or `.lex` file.

Notice, therefore, that the contents of these invocation files may be host-specific, in particular since the version number of the ACL system is embedded in it: the directory name `acl81` indicates Version 8.1 of the implementation. When a new Leonardo individual is installed it may therefore be necessary to edit invocation files in order to adjust them to the current host.

2.4 CLisp

CLisp is an open-source and implementation of CommonLisp that can be downloaded from the following URL.

```
http://www.clisp.org
```

The current version of Leonardo has been adapted for, and tested with CLisp running under Ubuntu Linux. We are not aware of incompatibilities with other versions of Linux, but obviously one is more on the safe side if one uses Ubuntu as well.

When Leonardo is run on CLisp then some of its facilities require the cURL program [1] to be installed on the same computer system so that it can be used by Leonardo, for example for operating the Session GUI, and for communicating with the Registrar. Please see the Technical Annex of the present report for further information.

[1] http://curl.haxx.se/

2.5 Other Implementations of CommonLisp

The Wikipedia page for CommonLisp [2] is a good source for general information about the language, and also for reference to major implementations. Another overview of implementations is found at [3] .

The primary ancestry tree for CommonLisp originates with MacLisp which was implemented at MIT Project Mac already in the 1960's and 1970's. After the standardization of Lisp dialects that resulted in CommonLisp it evolved into GNU Common Lisp (GCL) at MIT and into CMU Common Lisp (CMUCL). Notice however that GCL is not fully compliant with the standard.

The Steel Bank Common Lisp (SBCL) is a branch from CMUCL which claims to "be distinguished from CMU CL by a greater emphasis on maintainability."

Lispworks is another commercial implementation, besides Allegro CommonLisp, and with a lower price. It is produced by LispWorks Ltd. in the U.K.

There is also a number of other implementations as described on the cited webpage. As it often happens there are minor differences between the implementations, in spite of the fairly detailed standardization, which means that Leonardo does not immediately run on all of these implementations. The CLisp conversion was quite straightforward, however, so there is reason to believe that other conversions can also be done with very moderate effort.

There is a separate documentation report on Leonardo self-description and adaptation that provides necessary information for modification of the Leonardo system to run under additional varieties of Lisp.

[2] `http://en.wikipedia.org/wiki/Common_Lisp`
[3] `http://common-lisp.net/~dlw/LispSurvey.html`

Chapter 3

Basic Installation

The standard way of obtaining a Leonardo individual is to obtain a copy of an existing individual and to start using it, allowing it to reconfigure itself at the start of its first session. This is all that is needed in order to experiment with the Leonardo individual, for example, for trying the commands that were described in Chapter 1. This initial reconfiguration performs individualization which was described in Chapter 1.

Each computer host where a Leonardo individual executes must contain some information that describes the host and the services in it. The initialization of this information is referred to as *accomodation;* it takes place the first time that a Leonardo individual is run on the host in question. In particular, if a copy of a Leonardo individual is executed on a new host, then the session will start with the individualization and accomodation operations.

We use the term *basic installation* for the suite of individualization, accomodation, personalization and registration, and the broader term *installation* when incorporation of additional software into the individual is also included.

3.1 Overview of Basic Installation

For basic installation it is assumed that a copy of a Leonardo individual has been obtained, and that the appropriate invocation file has been adjusted so that it refers to the Lisp system at hand as described in Section 2.3. The four basic installation steps can then proceed as follows.

- *Individualization:* Start a session with the `remus` agent in the individual (or more generally, with its kernel agent). The software will recognize that a copy is being used, reset some of the information in it, and prompt the user for a *master password* that will then be used both within the new individual and for its communication with the Registrar.
- *Accomodation:* In a host that is new to Leonardo, this operation chooses a name for the host and a file directory where information about the host for the purposes of Leonardo will be stored.

- *Personalization:* This is the operation of adding information about the individual itself, its owner, and related information. It will normally be done in the initial session and just after individualization and accomodation (if applicable) have taken place, but it may also continue at later occasions.
- *Registration:* The primary purpose of the registration operation is to obtain a name for the present individual in the namespace of its registrar. At the same time, the Registrar returns up-to-date information about other individuals and computational entities in its name space.

There is more to it than this, however, and the following are the additional details that one needs to know about each of the steps.

3.1.1 Individualization

The basic installation process is started by starting a session with the `remus` agent in the individual (or more generally, with its kernel agent), using one of the invocation files that were described in Section 1.4. The software will recognize that a copy is being used, and not an original individual, and it will ask whether this is in fact the case. Provided that the user answers yes, the system deletes information that is specific to the individual that the copy was made from, and establishes the current copy as an individual. This is done as part of the session startup process, and there is no way of doing it as an invoked operation later on in the session.

Furthermore, and still during the session startup process, the user is prompted for a master password for the new individual. This password will be used for encrypting some protected information in the individual. There is no backup copy of the master password, so if the user forgets it then that protected information is permanently lost.

The reason why the system asks the user whether the individual at hand is indeed a copy, is as follows. Standard file systems assign a few timepoints to each file, including its time of creation, its time of latest update, and so forth. The time of creation is unchanged if the file is moved to a new location, but if a copy is made of the file then the copy obtains the new time of creation. This is at least the case for NTFS and some of the file systems used by Linux. Exploiting this feature, each individual contains one particular file that is only used for determining cloning and, in a distinguished location, it also contains a note of the time of creation of that file. If the entire individual is copied, then in the copy there is a mismatch between the actual time of creation and the noted time of creation for that file. This is how the startup process can determine that it is operating in a copy.

This method is not entirely perfect, however. If an individual is moved repeatedly between detachable memory devices then it may occur that the time of creation is reset although only move operations and no copy operation is performed. The same may happen if an individual is zipped, moved, and then unzipped. Therefore, if the system notices a change in the time of creation, it allows the user the possibility to state that the individual at hand is bona fide the original individual and not a clone, and in this case the system updates its note of the time of creation of the file being used for this mechanism.

It is also possible to cheat in the opposite sense, since it is in fact possible to reset the time of creation of a file. The system does not contain any mechanism for protection against such manipulation.

3.1.2 Accomodation

Whenever a Leonardo individual starts a session on a new computer host, ie. one where Leonardo has not executed before, there are two questions that must be resolved during the session startup process: the choice of name for this host for the purposes of Leonardo, and the choice of file directory where the self-description for the host will be stored. If the defaults are selected then the host name is chosen as the "computer name" according to the operating system, converted to lower-case letters, and prepended by the prefix `host.` For example, if the computer name is `Robin` then its default name for Leonardo will be `host.robin` .

Similarly, the defaults for self-description directories are

```
C:\Leohost\
~/Leohost/
```

in Windows systems and in Linux and Mac systems, respectively. These defaults may be overridden, but preferably during the initial startup session. The system will ask for the user's choice in these respects during the initial startup process.

These assignments constitute the first steps with respect to accomodation. Additional assignments will also be needed, for example concerning the owner of the host in question, and the location of various auxiliary softwares. These assignments can best be made using the Session GUI which will be described below.

If a Leonardo individual is located on a detachable memory device, eg. a USB memory stick, then the accomodation operation will also collect information about this device.

3.1.3 Personalization

A computational session with a Leonardo individual begins with a standardized startup sequence, and this is where individualization and initial accomodation take place when required. After this the session goes into a dialog with the *Command Line Executive,* CLE. It is possible to do personalization and registration using CLE commands, but it is generally recommended to use the *Session Graphic User Interface* (Session GUI) instead for these purposes. The Session GUI co-exists with the CLE, so during a session one can use the command-line dialog and the Session GUI alternatingly.

The Session GUI is started by entering the following CLE command:

```
loadk indiv-kb
```

This loads the knowledgeblock `indiv-kb` containing various services for self-organization of the individual, and if possible it also starts an internal web server within the session at hand, for use in localhost mode. Finally, it starts a web-browser window in a way whereby it accesses the Session GUI

for the present session. This provides a tool whereby the user can specify basic information for the new individual, such as who is its owner. If this is the first individual of the present user then the Session GUI can be used for introducing an entity representing this person. Also, if the presently used computer has not been used for running Leonardo systems before, then information about this host can be stated to the GUI as an aspect of the accomodation process.

The web server for the Session GUI is defined so that it only accepts input from the `localhost` client, ie. from within the same computer, and for obvious security reasons.

Aspects of the personalization operation are to be repeated when an individual obtains a new owner. Likewise, renewed accomodation is needed when the individual is started in another host that has not previously been visited by any Leonardo individiual.

In the Technical Annex of the present report we describe the measures that are to be taken if the Session GUI does not start, or if the web-browser does not appear.

3.1.4 Registration

Using the Session GUI, the user can select to register the new individual. The individual at hand does this by sending a message to the Registrar, more specifically, the same registrar as had previously registered the individual from which the copy was taken. The Registrar returns appropriate unique names for the individual and for its remus agent (ie. its kernel agent), and the individual reconfigures itself so as to self-describe in terms of these names. The Registrar also returns updated versions of entityfiles in `Indivmap` so that this structure will contain up-to-date information both about the individual at hand and about other recently registered individuals, users, and hosts.

3.2 Beyond Basic Installation

Besides the basic installation process, there are also operations for adding *external and internal facilities* to an individual, ie. additional software besides what was provided in the clone. Internal facilities are software modules that have been developed using the Leonardo architecture and that can be incorporated into an individual, whereas external facilities are conventional software that the Leonardo individual can invoke using interfaces. Text editors, web browsers, text formatters such as LaTeX, and database systems such as MySQL are examples of external facilities.

Internal facilities are represented as Leonardo knowledgeblocks whereas external ones are provided using installation scripts that perform some or all of the operations that are needed for obtaining the external facility. Knowledgeblocks for internal facilities and stubs for external ones may be obtained from an individual's registrar or from other individuals.

Some simple external facilities will be described below. More advanced external facilities as well as internal facilities are described in a separate

memo, *Facility Installation in Leonardo.*

3.3 Naming Conventions

The knowledge structures in Leonardo individuals are organized in the ways that were described in Chapter 1, that is, using KR-expressions where the elements are symbols, strings, or numbers. Symbols are used for representing *entities* in the application domain, and each entity may have values for a number of *attributes.*

Messages in the communication between individuals are also expressed as KR-expressions, which means that communication is greatly facilitated if different individuals use the same meaning for symbols, or at least for those symbols that may occur in individual-to-individual communication. To this end each Leonardo individual is member of a *leosociety* and each leosociety has a *registrar* that is responsible for the allocation of jointly used names for entities of various types.

A natural further design decision for the naming system is to define an *internal syntax* for names in such a way that the spaces of possible names for entities of different types are disjoint by construction. For example, the internal syntax for symbols representing dates may be of the form `yyyy-mm-dd` for example `2012-03-15,` and internal syntax for other types should be chosen in such a way that no namespace overlap may occur.

These design principles are also applicable for symbols that represent the entities that are involved in the communication system itself, such as the names for computer hosts, local area networks, humans that are the owners of Leonardo individuals and of hosts, and of course for the network-level names of the Leonardo individuals themselves. Moreover, in order to prepare for communication across leosociety boundaries it is natural to let the same naming conventions for these *network description entities* apply for all Leonardo individuals independently of leosociety membership.

The Leonardo user will encounter some network description entities during the personalization and accomodation steps, since the purpose of those steps is to provide information about (at least) users and hosts. The following information is therefore relevant for performing the personalization of a Leonardo individual.

The following are the major types that are used for network describing entities.

- Individuals, eg. `Pawn-17`
- Agents, e.g. `remus-77`
- Computer hosts, eg. `host.mylaptop`
- Detachable memory devices, eg. `dmd.my-usb-stick`
- Local area networks, eg. `lan.my-lan-at-home`
- Leonardo societies, eg. `soc.root`
- Users, eg. `/Larsson.Gunnar`

Individuals and agents have a dual naming system involving both simple names, as shown above, and global or "network-wide" identifiers that are composite entities. The latter will be described in the chapter on network access.

The internal syntax for most of these types uses a type code as a prefix for the "natural" identifier for the entity in question, as shown in these examples. This is not a universal practice in Leonardo, but it is used for network describing entities.

The idea with leosocieties is that if a person or group of persons wishes to use Leonardo individuals for some purpose, they can create a leosociety of their own so as to manage their own group of individuals and their own registration process, and define their own namespaces and naming conventions. For example, if a course leader wishes to provide each student in her course with his or her own Leonardo individual so that it can be used for doing course exercises, then it would be appropriate to introduce a separate leosociety for the course.

The registrar is used not only for registering software individuals, but also for registering hosts, memory devices, and the other types mentioned above. In doing so it is responsible for maintaining the uniqueness of names and for avoiding name collisions. In short, the registrar is the guardian of the leosociety's namespace, at least with respect to network description entities.

The leosociety registrar can also be used for application-specific purposes, such as keeping track of student results in the course example.

The naming conventions for persons are as follows. An entity of the form

```
Lastname.Firstname
```

possibly with a postfix for disambiguation, is used for characterizing generally known persons, for example

```
Einstein.Albert
```

This namespace is maintained by the registrar, or by a separate individual to which the registrar has delegated this task. At the same time, person-name entities of the form

```
/Lastname.Firstname
```

are used for representing persons in their role as network describing entities, ie. for persons that are directly involved with the operation of that leosociety, in particular as owners of individuals or hosts. Entities of this second kind have attributes that characterize the person's usage of the Leonardo individuals and of the services provided by them.

3.4 Details about the Installation Process

3.4.1 The Indiv-self and Indivmap Directories

One of the design principles for Leonardo has been that each Leonardo individual shall be entirely self-describing, so that all software and all data definitions that are needed for the operation of the individual shall be represented in forms whereby they are accessible for inspection and change by

the system itself, and using the same representation as is used for modelling the individual's environment. The basic system-oriented information that an individual has about itself is located in a directory called `Indiv-self` at the individual's top level, ie. sideways with the directories for the agents in the individual.

Furthermore, the basic information that an individual has about its computational environment, including other Leonardo individuals, computer hosts where these individuals are located, and so forth, is located in a similar top-level directory called `Indivmap`. The contents of this directory are administrated by a registrar and distributed to the individuals that it services.

Generally speaking, the purpose of the personalization operation is to update the contents of `Indiv-self` in the individual at hand, and the purpose of registration is to update the contents of `Indivmap` both in this individual and (eventually) in all other individuals in the same leosociety. Once in place, the information in these directories is used for a variety of purposes in the operation of the individual.

The contents of the `Indivmap` directory are organized as a knowledgeblock, called `indivmap-kb`. The contents of the `Indiv-self` is likewise organized as a knowledgeblock called `indiv-self-kb`.

3.4.2 Individualization

The example of the session prelude for a Leonardo session in Chapter 1 applied to an individual in original state. However, when the first session starts with an individual in copy state, then the session prelude is somewhat different since this is where the individualization operation takes place. The prelude may then appear as follows.

```
;   Loading C:\Leonardo\Copy of Pawn-74\remus\Startup\cl\bootfuns.leos

************************************************************
Now starting up a Leonardo session  --  please stand by

Welcome to a session with the Leonardo software individual Pawn-74
The session is done with the individual's agent called remus
This is session number 31 with this agent
This individual appears to be a copy, is this correct? y
Date when copy was made: 2012-09-07
This copy will now be individualized.
Please enter the master password for this individual: simsalabim
Thanks. Please assign personal attributes at earliest convenience.

This is an unregistered individual - use session GUI to register

This session executes on the following Lisp system:
International Allegro CL Enterprise Edition
8.1 [Windows] (Jul 28, 2007 7:42)
Copyright (C) 1985-2007, Franz Inc., Oakland, CA, USA.
   All Rights Reserved.

This development copy of Allegro CL is licensed to:
```

```
   [pc0756] Linkoping University

************************************************************
ses.001)
```

This shows that the startup process has noticed, at an early stage, that the software individual that is being started appears to be a copy. It asks the user for confirmation whether this is actually so, and when the user confirms this with a `y` or `yes` then the present individual is redefined as an individualized individual with reduced information about itself.

The major side-effect of this startup operation, ie. of individualization, is to update the contents of an entityfile called `indiv-descr` that is located in the `Indiv-self` directory. The first two entities in this entityfile must always be `indiv-descr` and `current-individual`. The following is an example of an entity description for the latter in an ordinary, *already registered* individual.

```
------------------------------------------------------
-- current-individual

[: type self-individual]
[: in-leosociety soc.root]
[: in-individual (leoind: Pawn soc.root 74)]
[: has-indiv-config Pawn]
[: directory-name Pawn-74]
[: created-on "2012-05-30"]
[: has-owner /Sandewall.Erik]
[: registration-state is-registered]
[: in-host host.dell-dc1
------------------------------------------------------
```

Its *individuality attributes* describe some aspects of the individual in question that are specific to it, and that are likely to differ between individuals. The attribute `directory-name` refers to the name of the individual, ie. how it may appear in a directory listing of individuals. This is usually, but not necessarily the name of the file-system directory containing the individual.

If this individual is moved to another location then only the `in-host` attribute may need to be reassigned, but there is no particular reason to reassign any of the others. However, if a copy of the individual is made then several attributes will be changed when the copy is individualized. The following is an example of the resulting new entity-description.

```
------------------------------------------------------
-- current-individual

[: type self-individual]
[: in-leosociety soc.root]
[: in-individual (leoind: Pawn soc.root 74)]
[: has-indiv-config Pawn]
[: created-on "2012-09-14"]
[: registration-state is-personalized]
[: old-directory-name Pawn-74]
[: old-has-owner /Sandewall.Erik]
[: in-host host.dell-dc1]
------------------------------------------------------
```

There is then a need for further operations that assign new values for those attributes that have been replaced. This occurs in the personalization and registration operations.

3.4.3 Personalization and Accomodation

The operations of personalization and accomodation will be discussed together since they can both best be performed using the Session GUI, but also using commands in the command-line dialog.

Describing the Current Individual

The following are the major operations that may be done during personalization and accomodation, and that concern the software individual per se:

- Assign the owner of the current individual to the `has-owner` attribute of `current-individual`
- Specify whether the location of the current individual is permanently with the host where the present session is running, or whether it is on a detachable memory device (dmd) that happens to be plugged into the present host. Introduce names for host and dmd if they do not already have names in the present leosociety.

In addition, the entities that are used for these attributes must be defined if they are not already known. An entity for a previously unknown user must be provided with an entity-description, and similarly for entities for a new host and a new detachable memory device. The description for a new host must include the information whether it is a stationary one or a detachable one. Usually this is the difference between a desktop on one hand, and a laptop or tablet on the other. This distinction is relevant if the hosts in question are used inside local area networks (LAN), since a detachable host may be attached to different LAN at different times. In the case of a LAN that is new and previously unknown to the current leosociety, this LAN must also be assigned a name and an entity-description.

All this information is established during the personalization and accomodation operations, and is transferred to the Registrar of the current leosociety during the registration operation. This transfer is not necessarily trivial since if name coincidence is detected then there is a need for interaction between registrar and user in order to clarify whether the same or different entities are intended, and to rename one of them if necessary.

Both the tentative and the final (registered) choice of name are represented explicitly in the individual, in the following ways. When new entities are introduced by the user during the personalization operation, they are added to the `indiv-descr` entityfile in `indiv-self-kb` which then serves as the workspace for the naming process. When the name has been accepted by the registrar for use in its leosociety, then its description is moved from `indiv-descr` to the appropriate entityfile in `indivmap-kb` such as `users-catal` or `hosts-catal,` depending on the type of the entity in question.

3.4.4 Registration

The final part of installation is to register the new individual with the Registrar of the leosociety at hand. When an individual is copied then the clone inherits the leosociety and, therefore, the registrar of the original individual. Moving an individual from one leosociety to another may be easy or difficult, depending on the extent to which there are differences between the organization of these leosocieties.

Registration is initiated by clicking the *Register* link in the Session GUI. This sends a request to the Registrar for obtaining a unique name for the present individual and for its kernel agent. The reconfiguration of the registering individual is performed automatically.

However, if additional entities were introduced in the course of personalization (usually using the Session GUI), then these entities must be uploaded to the Registrar so that it can add them to its catalogs of hosts, users, and other. The revised catalogs are then returned to the registering individual for insertion in its `Indivmap` directory and knowledgebase. When the updated `Indivmap` files have been received, the individual removes the newly-registered entities from its `indiv-descr` entityfile.

When entities of these kinds are uploaded it may occur that the Registrar is already using the same entity name as is being uploaded by the registrant. This may happen especially in off-line situations where individuals have not been reached by `Indivmap` downloads from the Registrar, so that different individuals have introduced the same name concurrently. In such cases the user must resolve the question whether it is the same actual entity that is intended in both places, or whether two different entities are intended. In the latter case the registrant and its user must change the name of the entity in question and try registering again.

The Session GUI provides the necessary rulebase and dialogs for registering such network description entities. In the case of computer hosts and dmd's it assumes for the purpose of default that no user assigns the same name to several hosts or several dmd's that he or she owns, whereas different users may do so. Coincidence of names can often be resolved on this basis.

The Registrar provides a directory name for the registered individual, for example `Pawn-39` or `Cappa-12.` This name is used for specifying counterpart individuals in individual-to-individual communication. It is natural, but not necessary, to also use it as the name of the top-level directory of the persistent manifestation of the individual in computer memory. Unless the user has been able to anticipate what the directory name will be, it will then be necessary to change the name of the currently used directory, after the registration has been performed, so as to conform with the assigned directory name.

3.5 Information Collected in Accomodation

3.5.1 Describing the Current Host

Leonardo individuals rely on some information that is specific to each host, and which is shared by all individuals that are located on, or visit the host

in question. (An individual may visit a host by being on a detachable memory device that is plugged into the host, or by being moved to the permanent memory of the host). Besides elementary facts such as host ownership and type of operating system [1] this includes information about what software facilities are available on the host in question. The purpose of the accomodation operation is to enter or update this information about a host.

In order to accomodate this information in a way whereby it can be shared between several resident and visiting individuals, there shall be a dedicated file directory called the `Leohost directory` whose normal location is at `C:/Leohost/` in Windows systems and at `~/Leohost/` in Linux and Mac systems. This directory shall at a minimum contain the following entityfiles:

```
leohost-kb
host-descr
facilities-catal
```

Additional entityfiles may be added in support of particular external facilities. The file `leohost-kb` contains a catalog of what files are in the structure, and their properties, as usual. The file `host-descr` contains a description of the host at hand, for example

```
------------------------------------------------------------
-- current-host

[: type host-self-type]
[: host-own-name host.dell-dc1]
[: has-owner /Sandewall.Erik]
[: has-host-type detachable-host]
[: has-allegro-path "C:\Progra~1\acl81\"]
------------------------------------------------------------
```

(The actual representation is slightly different in order to make it possible for the same host to have different attributes for different leosocieties, but this is rarely used).

When a session with a Leonardo individual is started on a previously unvisited host then the startup procedure will assign provisional values to some of these attributes. In particular, the hostname that is used by the operating system in question will be used as the value of the `host-own-name` attribute, although prepended by the type tag `host.` and converted to lowercase if applicable. This produces a proposed name for this host for the purposes of the current leosociety, but this proposed name may be changed by the user during the personalization process, namely, if the user prefers another name, or if he or she happens to know that another host that is used for the same leosociety is already using the proposed name. In fact, it may also be necessary to change the name during the registration step, namely, if it is determined in the registration dialog that the Registrar already uses this name for another host.

The values of the `has-owner` and `has-host-type` attributes obtain tentative values in similar ways, and these values may also be changed during personalization. The value of the `has-allegro-path` attribute, finally,

[1] Multiple operating systems on the same physical host are considered as separate hosts for the purposes of Leonardo.

specifies the location of the Allegro CommonLisp system in the host at hand. The startup process can usually find this itself, but in some situations it may be necessary to provide it manually. There is a similar attribute `has-clisp-path` that is used for the location of the CLisp system in the host. These attributes are used when particular aspects of the implementation need to be accessed.

Besides allowing the user to confirm or change the values of these attributes, the personalization process will also create a Leohost directory in the host if there is not one there already. The Session GUI page for the *Current host* will allow the user to confirm that the standard location will be used for the Leohost directory or to specify an alternative location.

There is in fact a chicken-and-egg problem when nonstandard choices are made which means that although it is possible to choose another hostname than the one based on the operating system's hostname, and it is also possible to choose a nonstandard location for the Leohost directory, it is not possible to use nonstandard choices for both. This is because each of them contains the information about the nonstandard choice of the other one, and if both are nonstandard then neither of them can be found.

The Session GUI allows the user to specify the values for these attributes. Since these values are stored in entityfiles, like virtually all information in a Leonardo individual, it is straightforward use them in application scripts and programs.

3.5.2 Describing Detachable Memory Devices

Each Leonardo individual shall have one specific *location* at any one time, although the location may change from time to time. The location may be specified as the name of a computer host, or as the name of a detachable memory device (dmd). In the latter case the individual can be moved from one host to another simply by attaching the dmd to different hosts at different times.

Each dmd that contains a Leonardo individual shall therefore have a name, for example `dmd.kingston-2` so that the location information can be represented. Many detachable memory devices make it possible to assign such a name to the device on the file-system level, and in these cases it is natural to use its file-system name as the basis for the Leonardo name in the obvious way, but this can be overridden during the personalization step in the same way as for host names.

The startup process for Leonardo sessions attempts to identify whether the current individual is presently located on a host or a dmd. This is simple in Linux and Mac environments but somewhat tricky in Windows environments, and user assistance via the Session GUI is sometimes required.

3.5.3 Describing Simple External Facilities

The host-specific information in the Leohost directory includes one more obligatory entityfile, namely `facilities-catal.` The following is an example of entities in this file:

```
-----------------------------------------------------------
-- facil.firefox
[: type os-command]
[: win-commandphrase "C:\Progra~1\Mozilla Firefox\firefox.exe "]
[: linux-commandphrase "/usr/bin/firefox "]
[: in-category facil.web-browser]
-----------------------------------------------------------
-- facil.iexplorer
[: type os-command]
[: win-commandphrase "C:\Progra~1\Internet Explorer\iexplore.exe "]
[: in-category facil.web-browser]
-----------------------------------------------------------
-- facil.notepad
[: type os-command]
[: win-commandphrase "C:\WINDOWS\system32\notepad.exe "]
[: in-category facil.texteditor]
-----------------------------------------------------------
-- facil.gedit
[: type os-command]
[: linux-commandphrase "/usr/bin/gedit "]
[: in-category facil.texteditor]
```

This information is used when the Leonardo individual is to invoke the external facility in question.

Each user has of course his or her preferences concerning which program is to be used for particular purposes, and therefore there is also an entityfile called `facility-preferences` that contains entity-descriptions like the following ones:

```
-----------------------------------------------------------
-- facil.texteditor
[: type facility-category]
[: linux-preference facil.gedit]
[: windows-preference facil.notepad]
-----------------------------------------------------------
-- facil.web-browser
[: type facility-category]
[: linux-preference facil.firefox]
[: windows-preference facil.firefox]
-----------------------------------------------------------
```

This means that each individual contains the software tools preferences for its owner. For situations where one user owns several Leonardo individuals, the Session GUI makes it possible for the user to maintain his personal `facility-preferences` file in the registrar, and to upload it to there and download it from there. The individual stores this file in its `indiv-self-kb` knowledgeblock.

Information of these kinds is normally entered during the original installation of an individual and using the Session GUI. If it needs to be changed at later times then this can also be done using the Session GUI. As an alternative it is possible as well to change it by editing the respective entityfiles directly, although caution must be exercised in order to respect the applicable data dependencies. Please refer to the detailed system documentation in this respect.

The external facilities in the examples have been described using a particularly simple interface, where the path to the program in question is combined with the path to the file or URL that the program shall operate on. Many programs require a more complex invocation method, with a command line containing several arguments or parameters, or even with two-way communication between the Leonardo individual and its external facility. The mechanism for such interfaces is still in a development state.

3.6 External Resources and Management of Passwords

The previous section described *external facilities* in the sense of other programs that run on the same host as the individual using them. There is also a provision for *external resources* in the sense of non-Leonardo-based softwares that are running on other hosts, so that they must be accessed using the Internet. This will be described here.

3.6.1 Password Management for External Resources

External resources usually require a username and a password for establishing a connection, and the following example shows how these are represented. It contains descriptions of two external resources, called `xr.intersango` and `xr.mtgox` . The value of the `with-password` attribute has been partly omitted.

```
-------------------------------------------------------
-- xr.intersango
[: type external-resource]
[: in-vault vault.green]
[: with-username erisa@ida.liu.se]
[: with-password <249 144 47 33 ... 231 93>]
-------------------------------------------------------
-- xr.mtgox
[: type external-resource]
[: in-vault vault.green]
[: with-username erisa]
[: with-password <98 1 113 222 ... 232 198>]
-------------------------------------------------------
```

The password is stored in encrypted form, and a separate key is required in order to decrypt the password so that the agent can open the connection. Passwords and other confidential information are figuratively considered to be stored in “vaults” and each vault has its own password which must be entered by the user anew in each session. In this way it is not necessary to enter passwords for specific external resources again in each session; they can be entered once and for all and stored in encrypted form. In the example, as soon as the user has entered the password for the Green vault in the current session, the agent is able to log into both `xr.intersango, xr.mtgox,` and any other external resource whose password is located in the Green vault, figuratively speaking.

The entityfile `access-info` is located in `indiv-self-kb` and is the recommended location for storing entities whose type is `external-resource`, such as those in the example. The Session GUI contains two pages that are relevant in this respect. The menu tab `This session` selects a page where the user may "open vaults" ie. enter the passwords for vaults so as to make their contents accessible. The standard configuration of Leonardo individuals uses three vaults which are characterized using the colors `orange`, `red`, and `green`. The orange vault password is the "master password" for the individual in question.

Moreover, the menu tab `External Resources` selects a page containing entries for the individual's external resources; it can be used for adding and removing such resources, and for changing their username and password.

Other information is also needed for external resources, beginning with one or more URL's that shall be used for accessing the resource. This information is however not stored in the `access-info` file since it pertains to the access methods, scripts and other information that is used for interacting with the resource. The username and password information is specific to each individual, whereas access methods and scripts may be stated once and for all and may be used by several individuals.

3.6.2 A Keyring for the User

Whereas the entityfile `access-info` contains password information that the Leonardo individual may use in its own computational processes, some users may wish to use the same facility for storing passwords that they use themselves for a variety of websites and other password-requiring situations. This service (sometimes called a keyring) is available under the `User GUI` tab in the Session GUI.

3.6.3 Specifying the Individual's E-mail Address

The Leonardo platform contains a `sendmail` facility, either as part of the Allegro CommonLisp implementation or as an external facility. This facility may be used for automatic mail purposes where the individual sends e-mail messages on behalf of its owner, but in cases where the outgoing email are not directly related to the owner it may be more natural to consider the Leonardo individual itself as the sender. This may be the case eg. for e-mail concerning a conference, or a university course. For these reasons the personalized individual may contain information both about its own e-mail address and the one of its owner. The former is represented using the `has-email` attribute of `current-host` and the latter using the `has-email` or `has-auto-email` attribute of the entity representing the owner.

The reason for the two attributes for the owner is that some users may wish to use a separate email account for his or her automatic outgoing email. If both attributes have been assigned a value then the value for `has-auto-email` is used by the Leonardo e-mail sender.

The passwords for these email accounts are stored using designated external resource entities in the `access-info` entityfile.

Chapter 4

Defining Action Verbs Using Scripts

Leonardo individuals are based on using KR-expressions as a textual representation of structured information. They are used for declaractive information, ie. descriptions of some phenomenon in the world, but they are also used for *scripts* ie. structured expressions that describe a way of doing something. The extensive definition of the syntax and the use of scripts is the topic of another report, but a general understanding of scripts in Leonardo is required background in order to read the next chapter which addresses installation aspects of servers and message-passing, as well as for a subsequent report describing and external and internal facilities. The present chapter is intended to provide such an overview.

4.1 Major Scripting Languages in Leonardo

There are several distinct scripting languages in Leonardo, distinguished by the purpose they serve. The two major ones are the *Action Scripting Language* (ASL) that is used for defining composite actions that may be performed by a Leonardo agent during one of its sessions, and the *Document Scripting Language* (DSL) that is used for producing short and long segments of text, ranging from brief error messages to dynamic webpages. The closely related *Text Scripting Language* (TSL) is a markup language that is used for the source text in reports and articles (including the present report).

Both the ASL and the DSL use KR-expressions as their syntactic form and they also share many of the semantic conventions, but they use different sets of primitive operations. The following are some very simple examples by way of introduction. Please recall that the primitive operation `put` with three arguments was introduced in Chapter 1 as an operation that assigns a value to a given combination of carrier entity and attribute. The following is an example of an ASL script that performs several such put operations.

```
[soact [put .e type fruit]
       [put .e color .c]
       [put .e climate-zone .z]
```

```
[put fruits-catal contents
     (cons .e (get fruits-catal contents)) ]]
```

If this script is called `addfruit` then the following command

```
[addfruit :e banana :c yellow :z tropical]
```

will add the entity `banana` to the sequence that is the value of the `contents` attribute for the entity `fruits-catal` and it will also assign values to `banana` for the three other attributes that are mentioned in the script.

One can see from this example that variables are represented using a symbol that is preceded by a full stop character, and that when a parameter tag (symbol preceded by a colon character) occurs in the command, then the corresponding variable (colon replaced by full stop) is bound to the parameter value for the purpose of evaluating the script. The operator `soact` stands for "sequence of actions."

By way of comparison, the following is a simple example of a DSL script that may use data that have been assigned by the `addfruit` script.

```
[fragment "The color of" [e .e] "is" (get .e color)
    "and it thrives in the" (get .e climate-zone) "zone." ]
```

If this script is evaluated in a context where `.e` is bound to `banana` then it may produce the following text in HTML encoding for use in a web server:

```
The color of <em>banana</em> is yellow and it thrives in
the tropical zone.
```

and it may also produce LaTeX code that in turn produces the following in a pdf context:

The color of *banana* is yellow and it thrives in the tropical zone.

In other words, DSL scripts may be used for generating HTML markup, LaTeX markup for further processing to formatted pdf text, and also a plaintext format for use eg. in the command-line dialog. A variety of formatting operations are defined as DSL operations, functional expressions that access the current information state in the agent at hand may be used, and conventional statements for conditional expressions, repetition over a sequence and set, and other similar control constructs are also available.

Both these simple examples of scripts contained *terms* ie. expressions that are evaluated in order to obtain a value that is used in the further processing of the script. Terms are formed recursively and are surrounded by round parentheses, and the same syntax for terms is used in both ASL and DSL.

It is obvious that a software platform such as Leonardo must have a scripting language for defining composite commands, but it may not be as obvious that a document scripting language is a basic necessity. However, in fact it is very useful to have the DSL as a basic construct. It is used for defining the dynamic web pages in the Session GUI, which is a basic resource for the system as we have seen. It is also sometimes used for defining the responses when commands are entered in the command-line dialog. The relation between the two scripting languages is reciprocal since the ASL may be used for defining the effects of dynamic web pages, although in fact these effects are often defined on the Lisp level in the current system.

4.2 Scripts that Define Actions

4.2.1 Script Definitions in Entityfiles

Each script must have a name, and the obvious way of arranging this in the Leonardo system design is to consider the script name as an entity and to assign the script expression to it as an attribute value. This is possible but impractical for a number of reasons. Script definitions are therefore written as *properties* of their names when they appear in entityfiles, like in the following example.

```
------------------------------------------------------
-- addfruit
[: type leoverb]

@Performdef
[soact [put .e type fruit]
       [put .e color .c]
       [put .e climate-zone .z]
       [put fruits-catal contents
            (cons .e (get fruits-catal contents)) ]]
------------------------------------------------------
```

Properties are similar to attributes in Leonardo, but they are always strings, and usually multiline strings, and they may be used for scripts, segments of Lisp code, pieces of text eg. for documentation purposes, or any other use where long strings are needed. Each property assignment begins with a line containing the at-sign `@` followed by the name of the property. The following lines are interpreted as the property value, up to the next line that begins with an at-sign or to the line of dashes that separates entity descriptions. There is an informal rule to distinguish attribute names and property names by capitalizing the first character in the latter.

In order to define a script for a Leonardo individual, one shall therefore define it as a property of an entity in the way shown in this example. One additional step must be taken: the leading entity of the entityfile where the definition is placed must have the following attribute assignment.

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

For example, if the definition of `addfruit` occurs in an entityfile called `fruits-catal` then the first entity-description in that entityfile might be

```
------------------------------------------------------
-- fruits-catal
[: type entityfile]
[: in-categories {verbfile}]
[: contents <fruits-catal addfruit remfruit amendfruit>]
------------------------------------------------------
```

The `in-categories` attribute is used in general for associating a set of "flags" with an entity; flags are similar to attributes except they have no associated value, but merely presence or absence. The effect of including the `verbfile` flag with an entityfile is that after it has been loaded or reloaded into a session with a Leonardo individual, the loaded entityfile contents are processed and in particular each `Performdef` property is parsed

and converted to an internal datastructure of the same kind as is used for attributes. In principle, the `Performdef` property becomes a `performdef` attribute.

4.2.2 Execution of Action Scripts

The most obvious and immediate use of action scripts is to use them as higher-level commands in the command-line dialog. Scripts can also 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.

4.2.3 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, but also in the definition of dynamic web pages and in the definition of message-passing between agents. For example, the `addfruit` command in the example can also be invoked from a browser using the following URL:

```
http://localhost/addfruit?e=banana&c=yellow&z=tropical
```

where the use of `localhost` is of course just an example. Moreover it is possible for one Leonardo agent to send a command for execution in another agent using an action such as

```
[send-to-agent fruitserver
       [addfruit :e banana :c yellow :z tropical] ]
```

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 such 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 one agent 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
     [if [-equal .c red][put .p has-color .c]]
     [set-outcome (if [equal .c red]
                      [fail: no-red-color "Cannot assign red 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 except if the required color is red. It returns a `fail:` outcome if the color is red and a `result:` outcome otherwise, using the verb `set-outcome` . The `-equal` symbol represents the negation of the `equal` predicate.

4.2.4 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 agent 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 Symbol :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 agent 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 is written in the Document Scripting Language and specifies what HTML contents are to be sent back for a given outcome record from the performance definition. More about this later.

4.2.5 Definition of Parameter Types

One additional 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:

```
-------------------------------------------------------
-- addfruit
[: type leoverb]
[: paramtypes [def :e Symbol :c Symbol :z 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.

4.2.6 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 agent, `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 Symbol :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 Symbol :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.

4.3 Scripts that Define Output Text

Since the `Performdef` definition only specifies how an action is to be performed, and not how the results are to be displayed, there is a need for one or more additional definitions. The case where an action is requested from the command-line dialog requires one kind of definition; the case where it

is requested from the built-in web server due to an incoming request from a web browser requires another kind of definition. These cases are distinguished using different properties. In this section we shall only describe the case of web-server definitions since it is the most illustrative one.

4.3.1 A Simple Example

The following example extends the `addfruit` example above with a simple webpage response. It is assumed that the server request has the same form as before, ie.

```
http://localhost/addfruit?e=banana&c=yellow&z=tropical
```

The accessed website could contain the following definition.

```
-----------------------------------------------------------
-- addfruit
[: type leoverb]
[: paramtypes [def :e Symbol :c Symbol :z Symbol]

@Performdef
[soact [put .e type fruit]
       [put .e color .c]
       [put .e climate-zone .z]
       [put fruits-catal contents
            (cons .e (get fruits-catal contents)) ]
       [writefil fruits-catal]
       [set-outcome [result: "OK" :e .e]] ]

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

The `Displaydef` definition is written in the Document Scripting Language. Notice that the `Performdef` definition has been amended with two more sub-expressions, compared to its earlier definition for this example. First, `[writefil fruits-catal]` 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 `e` parameter. The importance of this is that the `Displaydef` definition is executed in a context where the variable bindings are those that are obtained from the outcome of the `Performdef` definition, which is why the variable `.e` can be used in the `Displaydef` in the example.

4.3.2 The Text Scripting Language

It may be mentioned in passing that besides the Document Scripting Language there is an alternative Text Scripting Language (TSL) that is used for preparing reports, articles and other non-interactive documents. The DSL and the TSL have different syntax but are parsed into the same internal representation. Consider the following pair of examples.

```
DSL:   [fragment "An example of using" [b "boldface"] "and"
           [e "italic"] "fonts."]
TSL:   An example of using [b boldface] and [e italic] fonts.
```

Therefore DSL is adapted to situations like the specification of dynamic webpage contents where there is a need to specify input fields, tables of rows and columns, repetition of fields, and so forth. This expressivity is obtained at the expense of a certain syntactic overhead compared to TSL which essentially uses only two types of reserved characters: square brackets for delimiting subexpressions that are led by a formatting operator, as in the example, and a colon at the beginning of a word which is used for specifying parameters. The following example illustrates the latter aspect.

```
TSL:   [heading Technical Details :level 2]
```

It is straightforward to embed DSL source within TSL and vice versa since they are parsed to the same internal representation. Generation routines for HTML, LaTeX and plain text are defined to apply on that internal representation.

This compact and unified architecture for commands and texts of various kinds has many advantages. For example, it makes it possible and convenient to produce system documentation as pdf files where the descriptions of individual command verbs and other primitives are stored in the knowledgebase of a Leonardo individual so that they are compiled into a finished document when the mixed TSL/DSL source is converted to LaTeX, and from there to pdf.

Chapter 5

Scripts Defining Webpages and Communication between Agents

5.1 Dynamic Web Pages

5.1.1 Input Forms

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

```
http://localhost/addfruit-specify
```

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

```
------------------------------------------------------------
-- addfruit
[: type leoverb]
[: paramtypes [def :e Symbol :c Symbol :z Symbol]

@Specifydef
[body [heading "Entry of Fruit Information" :level 3]
      [request :to addfruit :method post ^
          [table :frame box :rules all :border 1 ^
              [row [box "Name:"]
                   [box :charsize 50 ^
                        [input text :tag e] ]]
              [row [box "Color:"]
                   [box :charsize 50 ^
                        [input text :tag c] ]]
              [row [box "Climate zone:"]
                   [box :charsize 50 ^
                        [input text :tag z] ]] ]
```

```
          [sendbutton "Enter"]
               ]]

@Performdef
[soact [put .e type fruit]
       [put .e color .c]
       [put .e climate-zone .z]
       [put fruits-catal contents
            (cons .e (get fruits-catal contents)) ]
       [writefil fruits-catal]
       [set-outcome [result: "OK" :e .e]] ]

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

The `Specifydef` definition uses the Document Scripting Language (DSL) like the `Displaydef` definition. It is structurally similar to HTML and should be self-explanatory to anyone who knows HTML, although some of the antimnemonic 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 expressive and 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 `Specifydef` defines a webpage containing a table containing three data input fields and a button labelled `Enter.` If the input fields are filled with `banana, yellow,` and `tropical,` corresponding to the examples in the previous chapter, then the following URL will be constructed when the `Enter` button is clicked.

```
http://localhost/addfruit?e=banana&c=yellow&z=tropical
```

and this will then cause the `Performdef` and after it the `Displaydef` to be performed.

This example 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 `addfruit` , the first webpage definition is attached to the identifier `addfruit-specify` and the second one as well as the `Performdef` is attached to the identifier `addfruit` . This is why the page containing the form is accessed as `http://localhost/addfruit-specify` , and it is also why the definition of the form specifies `addfruit` 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.

5.1.2 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 agent 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]
[: paramtypes [def :e Symbol :c Symbol :z Symbol]

@Performdef
[soact [put .e type fruit]
       [put .e color .c]
       [put .e climate-zone .z]
       [addmember (get fruits-catal contents) .e]
       [writefil fruits-catal]
       [set-outcome [result: :type fruit-report :value .e
                     :in-countries (get .e exported-from)] ]]

@Displaydef
[body [heading "Thank you" :level 3]
      "Your contribution of valuable information about" .value
      "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 .e 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.

5.2 Script Invocation and Repetition Statements

The DSL language in analogous to ASL in many ways: it allows one script to invoke another one as a "subroutine" and it contains control primitives such as conditionals and loops. A brief example will illustrate this using the `enumerate-list` operation that was used in the `Displaydef` above. The example assumes that the value of e.g. `(get banana 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 following is a very simpleminded definition for doing this.

```
-----------------------------------------------------------
-- enumerate-list
[: type leoverb]

@Displaydef
[table
    [repeat c .list
        [row [box .c]] ]]
-----------------------------------------------------------
```

This definition will simply construct a table consisting of one row for each element in the sequence given as the `.list` parameter, where each row consists of only one cell, containing the element in question.

This example illustrates one of the two ways that a DSL script can invoke a DSL-defined verb, namely, as a macro that constructs a subexpression in the HTML page (for example) being constructed. The other way is when the invoked verb is a separate command word for the web server. Continuing with the same example, the `Displaydef` for `newfruit` and the definition for `enumerate-list` might instead be written as follows.

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

@Displaydef
[body [heading "Thank you" :level 3]
      "Your contribution of valuable information about" .value
      "is greatly appreciated by our website."
      "The major exporting countries of this fruit are shown"
      [link "here." :path [enumerate-list :list (url-encode .in-countries)]] ]
-----------------------------------------------------------
-- enumerate-list
[: type leoverb]

@Displaydef
[table
    [repeat c (url-decode .list)
        [row [box .c]] ]]
-----------------------------------------------------------
```

In this case the `Displaydef` will produce a page where the word `here` is a clickable link of the form

```
http://localhost/enumerate-list?list=...
```

where the `list` parameter contains an encoding of the set that was obtained as the value of `(get .e exported-from)`. The encoding and decoding is required since a set of symbols in Leonardo notation is not suitable for being included as a parameter in a URL.

Notice therefore that the necessity to encode composite expressions when they are used as parameters for DSL verbs applies only when the invocation

is done by way of a clickable URL link, and not when it is done by way of macro expansion.

5.3 Agent-to-Agent Communication

In the particular case of message-passing between agents the presentation script is irrelevant on the server side, 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.

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 .e exported-from)]
```

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

Notice 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.

5.4 Leonardo Dynamic Resources

We have now described how definitions may be written whereby a Leonardo agent can operate as a server in agent-to-agent communication and as a web server. Since each agent can act in several roles of this kind it has been necessary to introduce a structure that defines which agents can communicate with which other agents, and using what types of commands. The present section describes this structure, which is referred to as the structure of *dynamic resources* in Leonardo.

The technical background for this is that each Leonardo individual contains the software for operating a web server; this web server is used both as a server for web browsers and as a server for communication between individuals. The webserver subsystem makes it possible to operate several separate logical servers that are distinguished by serving separate port numbers.

5.4.1 Allocation of Port Numbers

Although each particular interaction is arranged in terms of a client and a server, there will be applications where a group of individuals will communicate in arbitrary combinations, each one sometimes acting as a server and sometimes as a client. The Leonardo structure for servers and agent-to-agent communication is therefore organized as follows. Each leosociety (cf. Section 3.2) defines a number of individually named *server resources,* for example `dres.base` and for each agent in each individual it is specified what are the resources where it participates as both server and client, and what are the additional resources where it only participates as a client.

In fact one may think of a server resource as a subnet where a group of Leonardo agents may communicate with each other, and where some additional agents may send requests to the subnet members.

In those cases where an agent participates as server in one or more such resources, it needs to use separate port numbers for these so that the message traffic is not confused. Moreover, if several agents are active as servers at the same time on the same computer host then care must be taken so that there is no conflict with respect to port numbers. This applies both if these agents are in the same individual, and when they are in different individuals.

One possible way of arranging this might have been to have a port-number server and to distribute port numbers dynamically, and then to broadcast the assigned port numbers to participating agents. However, the present implementation uses a simpler solution, where the port number for a particular server agent in a particular server resource is calculated as the sum of two numbers, namely, a *base number* which is defined for each dynamic resource (ie. each "subnet") and an *increment* that is specific to each agent. Both base numbers are increments are specified as attributes in the relevant catalogs in `indivmap-kb`.

5.4.2 Requests in Agent-to-Agent Communication

Agent-to-agent communication in Leonardo is performed using the same internal web server as is used when an agent serves a web browser. Requests are therefore considered in principle as Leonardo command expressions which are however converted to URL requests as a way of sending them to the server side.

Continuing with one of the examples in the previous chapter, if an agent wishes to send the Leonardo-level command

```
[addfruit :e banana :c yellow :z tropical]
```

as a request to another agent that is able to listen to this using (one of) its webserver(s), then the client agent could issue the request as follows:

```
http://localhost/addfruit?e=banana&c=yellow&z=tropical&a2a=t
```

This is just like the request that would be sent from a web browser, as discussed above, except that the `a2a` parameter indicates that the client is a Leonardo agent and not a web browser, so that the outcome of the

`Performdef` shall be returned as is, and not the formatted expression according to the `Displaydef.`

Normally such a Leonardo-level command is used in a context that defines which agent the command shall be sent to.

Because of the structure of URL requests, in each Leonardo-level command that is used in this way the arguments and the parameter values should be single symbols, strings or numbers, and the use of composite expressions is not possible except in particular cases. For those cases where larger or structured data are to be transmitted in the request, there are two possibilities: either to use an encoding and decoding operation as described above, or to include an *attachment* with the request, similar to an e-mail attachment. The attachment may contain arbitrary text or binary data, and it will then be up to the server to parse its contents if necessary. Only one attachment can be included with a request of this kind.

In applications involving agent-to-agent communication it is usually the case that the message is composed and expedited either using Leonardo-level scripts of the kind described in Chapter 4, or using specialized Lisp code. However, the sending of requests directly from the command-line dialog is also useful, for example in debugging and software development situations.

5.4.3 Associating Verbs with Dynamic Resources

Since each server resource has the character of a subnet of Leonardo individuals it is natural to consider each request verb, such as `addfruit` in the example above, as belonging to one specific server resource. In fact this applies both to verbs that are used for defining web pages for use by web browsers, and to verbs that are used for agent-to-agent requests. In fact, it is often possible to use one and the same verb for both purposes, merely distinguishing the agent-to-agent usage by the `a2a` parameter as in the example above.

For these reasons each verb that is used for webserver or message-passing purposes must have a value for the attribute `in-dyn-resource` like in the following example

```
-------------------------------------------------------
-- newfruit
[: type leoverb]
[: in-dyn-resource dres.base]
[: paramtypes [def :e Symbol :c Symbol :z Symbol]
-------------------------------------------------------
```

The dynamic resource `dres.base` is the one that is used for the Session GUI.

The web server contains a mechanism for associating session identifier entities and passwords with particular dynamic resources. In those cases where a dynamic resource is declared to use a password protection but one particular command verb within the resource is an exception and should be available without password (for example, the login page) then this is declared as in the following example.

```
-----------------------------------------------------------
-- newfruit
[: type leoverb]
[: in-categories {no-password}]
[: in-dyn-resource dres.base]
[: paramtypes [def :e Symbol :c Symbol :z Symbol]
-----------------------------------------------------------
```

These declarations are only needed for verbs that are used directly in network-level communication, so that they can appear as the command part (succeeded by the question mark) in dynamic HTTP requests. They are not needed for verbs that are used as macros inside DSL scripts.

5.5 External Dynamic Resources

Besides Leonardo Dynamic Resources which were described above, there is also a notion of *external dynamic resources* ie. non-Leonardo web servers of any kind for which a Leonardo agent has an interface routine so that it can send requests to the external resource and parse the returned result into structured data for its own use. External dynamic resources are listed in the `Indivmap` file `foreign-servers-catal` in the same way as the Leonardo-specific ones, for example

```
-----------------------------------------------------------
-- fserv.sics

[: type foreign-server]
[: has-own-url "http://www.sics.se"]
-----------------------------------------------------------
```

There is a library routine for converting HTML and XML output from the external resource to KR-expressions and some auxiliary routines for managing such data. Beyond this there is no standardized facility for the access routines to such external dynamic resources.

Chapter 6

Annex: Technical Details

An Annex for Technical Details pertaining to the material in the present report is published as a separate document. Due to the character of its contents it is likely to undergo more frequent revisions than the present main report.