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:	2013-01-02

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 properties of a Leonardo system instance is that it is *time-aware, action-aware,* and *communicable.* Its knowledge representation system includes explicit representation of actions, including both its own actions, the actions of other Leonardo instances, and the actions of its user(s). Calendar and clock time is also represented. 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 is 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 all of 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. Being registered is the normal state.

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.

[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/`

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

- Remove data that pertain to the history of the individual that was copied.
- Ask 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, for example for querying other individuals and delegating actions to them.

An individual that has been obtained as a copy of another individual and that has been 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 in Chapter 3). These names are formed as consisting of a species name and a serial number, for example `Curator-4` for one individual in the `Curator` 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 `Novice` species contains one agent of each of the `remus, demus,` and `utilus` varieties. `Novice` individuals have an elementary character and they are used for initial exercises and tests by new users.

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 `Novice` 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. `Novice-4.` 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 `Agent-self` and a path such as

```
Leonardo/Novice-4/demus/Agent-self/main/
```

leading to the *invocation directory* for the agent in question. This is the directory level where computations with this agent will often 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 Common-Lisp. 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.

Another method of starting a session is to right-click the icon representing the individual and selecting the intended agent from the context menu that appears. This requires the definition of an additional script, however, and will be described in Chapter 4.

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\Novice-4\demus\Startup\cl\bootfuns.leos

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

Welcome to a session with the Leonardo software individual Novice-4
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 following file, in the example,

```
Leonardo/Novice-4/demus/Agent-self/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.

The arguments of the command verbs can be arbitrary CEL terms. Users of Lisp should notice that symbols are not evaluated and there is no separate quote operator. Instead there is a separate notation for variables, as used in the following example.

```
ssv .g green
put a color .g
```

where `.g` is a variable that is assigned the value `green` and then in used as the third argument of `put,` in the example. This is analogous to how variables are preprended by a question mark, for example as `?g` in some other AI languages. The symbol `ssv` is an acronym for *set session variable* since the assignment is not preserved to later sessions.

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. There is one exception, since the negation of the equality relation is written as `/=` and not as `-=` .

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

```
exit
```

Just terminating the session job does not have any major detrimental side-effects. One result is however that the agent history will not be able to record the ending-time of the session. Agent history is described in Section 4.2.3 in Chapter 4.

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 textual 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/Novice-4/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

The example in Section 1.4 makes use of a file with the extension `.leos` whereas the standard extension for entityfiles is `.leo` – a few words of explanation for this are 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\Novice-4\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. Allegro CommonLisp is the one that has been used for developing Leonardo, but 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 and CLisp 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 was done for an earlier version of the system but it is no longer 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` .

The second operation can also be made simply by drag-and-dropping the file `allegro-express.exe` to a copy that is then renamed as `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/Novice-4/demus/Agent-self/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 implementation 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
```

or obtained as follows in major versions of Linux

```
sudo apt-get install clisp
```

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.

Once it has been installed, the Clisp interpreter can be invoked using

```
clisp -modern
```

when Modern Base Lisp is desired.

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.

2.5 Other Implementations of CommonLisp

The Wikipedia page for CommonLisp [2] is a good source of 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.

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

[2] `http://en.wikipedia.org/wiki/Common_Lisp`

[3] `http://common-lisp.net/~dlw/LispSurvey.html`

Chapter 3

Basic Installation

We use the term *basic installation* for the operations that are needed in order to start using a Leonardo individual in a correct way on a computer host, and the broader term *installation* when incorporation of additional software into the individual is also included. Basic installation therefore includes the operations of individualization, accomodation, personalization and registration, which were described in Chapter 1. The present chapter describes and explains basic installation.

3.1 Leonardo Home Directories

The normal arrangement in a computer that accomodates one or more Leonardo individuals is that there are at least two directories of interest, one usually called `Leonardo` and one usually called `Leohost.` The Leonardo directory contains the Leonardo individuals as immediate subdirectories. When such individuals are mobile then they are moved into, or out of the Leonardo directory. The Leohost directory, on the other hand, contains descriptions of some aspects of the host at hand. In particular, it documents the existence and location of softwares that may be used by the Leonardo individual, such as text editors and web browsers.

The default location for the Leohost directory is as `C:/Leohost/` under Windows and as `~/Leohost/` under Linux. The Leonardo directory may be located on any level, but it is suggested to keep it on the top level as well, for example as `C:/Leonardo/` [1] .

Leonardo directories may be used in detachable memory devices such as USB memory sticks, and not merely in the stationary file structures of computers. Leohost directories are only used in the latter.

If copies of individuals are made for backup then it is recommended to distinguish them by using a separate directory for the copies, for example as `C:/Leonardo-copies/`, and not by changing the name of the copy.

[1] CommonLisp allows the use of the `/` character internally as the level separator in filepaths, regardless of whether a Unix-based or a Windows-family operating system is being used, and we shall follow that practice here.

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, and then to register itself. To this end it is sufficient to create a Leonardo directory, place the individual inside it, and invoke it in the way that was already described in Chapter 1. The Leohost directory, on the other hand, can be obtained in either of two ways:

- Start the session with the current individual, and issue the command `creleo` without arguments. This has the effect of creating a Leohost directory with initial contents.
- Obtain a copy of a Leohost directory from elsewhere and put it in the standard location of the host at hand.

However, in either case it is necessary to edit some of the files of this Leohost directory and adjust their contents so that they correctly describe the properties of the host at hand, and the relevant software resources in it. In particular, this must be done in order to be able to start the individual's Session GUI, which provides a graphic interface to the individual.

Once a Leohost directory has been set up, it can be used by all Leonardo individuals that reside in or visit the host in question. For example, if an individual is kept in a USB memory stick which is used in different hosts at different times, then in each particular session it will use the Leohost directory of the host being visited in order to obtain a model of its immediate hosting environment. Visiting individuals can of course also update the contents of the Leohost directory. The creation and use of a Leohost directory by an individual is what was referred to as *accomodation* above.

3.2 Steps to Session GUI and Registration

There are two major ways of interacting with a Leonardo individual, namely, through the Command-Line Executive (CLE) and through the Session GUI. Many operations can only be done using the CLE, but the Session GUI is more convenient in those situations where it can be used, and it can be extended to fit additional needs and new applications. The following are the steps that must be taken in order to bring an individual to the point where it can use the Session GUI.

- Obtain a copy of a Leonardo individual that you can have for your own use.
- Place it in a Leonardo directory as described above.
- Start a session with the kernel agent of this individual (details below). This has the effect of immediate *individualization:* it becomes an individual in its own right. It then starts the CLE command-loop and allows the user to issue commands.
- Generate a Leohost directory by issuing a `creleo` command as just described.
- Edit the two entityfiles in this Leohost directory, and load the corrected files into the session. In particular, verify that there is a correct specification of the path to a web browser in the host at hand. This

is an operation of *accomodation* whereby the individual "makes itself comfortable" in this host. See Section 3.7 for additional details.

- Start the Session GUI by issuing the `stagui` command to the CLE. This will start a web browser in a way where it accesses the built-in web server in the Leonardo individual, in `localhost` mode, and with dynamic web pages whereby one can modify the contents and settings of the individual.
- Use the Session GUI for reviewing and setting information about the individual at hand and the host at hand, for example, for setting who is the owner of the individual and the computer host. This is the operation of *personalization.* The session GUI can also be used for continuing the accomodation process.
- Use the Session GUI for sending a registration request to the applicable Leonardo Registrar. This is the operation of *registration.* Provided that it is successful, it will return a network level name for the present individual, together with some other information for its use.

It is recommended to then do the following at once:

- Terminate the session at hand, and rename the main directory of the individual at hand (ie. its immediate subdirectory of the Leonardo directory) so as to agree with the individual's name as assigned by the Registrar.

The above is the normal sequence of steps to be taken, but some variations are also possible. As already mentioned, it is possible to obtain the Leohost directory by copying it from another place; it may also have been included in the distribution obtained. Also, the registration operation may be performed using a command to the CLE, and not necessarily using the GUI.

This is all that is needed in order to start working with a Leonardo individual, for example, for experimenting with the commands that were described in Chapter 1.

3.3 Starting a Leonardo Session

3.3.1 Starting a Regular Session

There are two major ways of starting a session with an agent in a Leonardo individual. The "primitive" method which was described before is to navigate to a directory level inside the individual (seen as a directory and file structure) which has the form eg.

```
.../Leonardo/Novice-4/remus/Agent-self/main/
```

if the individual is `Novice-4` and one wishes to start a session with its kernel agent, which is normally called `remus` . This *invocation directory* contains *invocation files,* such as

```
winleo.bat
acleo.lex
```

for use in Windows and Linux, respectively. The session is started by clicking the `.bat` file, or by doing

```
./acleo.lex
```

in Linux. This is the only method that works when no Leohost directory is present, for example for starting the first session in the sequence of steps above.

The other method for starting a session consists of right-clicking the icon for the individual and selecting the appropriate operation from the 'context' menue obtained. This has presently only been implemented for Windows, and requires two things:

- The introduction of an appropriate entry in the "Send to" submenu of the set of options being obtained as the result of right-clicking the icon of the individual.
- The presence of a matching file in the Leohost directory. Default Leohost directories contain a file called `winleo-remus.cl` that is used for starting a session with the remus agent of an arbitrary Leonardo individual.

The user is therefore asked to accept the relative clumsiness of the first method since after only a few steps it will be possible to also use the second method.

The effect of the first method is that the computational session will start in the invocation directory. This causes a problem in those Linux systems that will not start a session from a detachable memory device, since one may sometimes wish to store individuals on such devices. For such cases there is a possibility of starting a session using an invocation file in the host's Leohost directory, and in such a way that it will anyway be a session for the individual in the detachable memory.

In all these cases, please recall that it is only possible to have one session at a time with a given individual, unless the setting that prevents duplicate sessions is explicitly turned off.

3.3.2 Starting an Inaugural Session

The first session with a newly made copy of a Leonardo individual is called an *inaugural session.* This is where the individualization step takes place, as described in Session 3.2. Such inaugural sessions must always be made with the kernel agent, i.e. with the `remus` agent. Moreover, if the host at hand is new from the Leonardo point of view, so that there is not yet any `Leohost` directory in it, then the method of invocation files must be used for starting the inaugural session.

Chapter 1 contained an example of the initial messages in the command-line window when a session is started. This example referred to the startup of a normal session. The message sequence that is obtained during startup of an inaugural session is somewhat different, as the following example shows.

```
;   Loading C:\Leonardo\Copy of Novice-4\remus\Startup\cl\bootfuns.leos

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

Welcome to a session with the Leonardo software individual Novice-4
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 system then asks the user to provide a *master password* for this new individual. It is important to keep a note of what password was selected, since the system does not provide any ways of recovering it if it is lost.

3.4 Detailed Aspects of the Basic Installation Steps

This section contains additional details about the basic installation operations, i.e. individualization, accomodation, personalization and registration. It may be skipped on a first reading of the report.

3.4.1 Individualization

The basic installation process is started by invoking 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 gives a yes

answer, 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 will be 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. Currently used 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. Exploiting this feature, each individual contains one particular file that is only used for determining cloning. In a distinguished location the individual also stores 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 mislead the system 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.

3.4.2 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 as already described.

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.

3.4.3 Personalization

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:

```
stagui
```

This loads the knowledgeblocks `indiv-kb` and `sesgui-kb`. The first of these contains various services for self-organization of the individual, including those invoked by the Session GUI, and the latter one contains the scripts for the GUI webpages. 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.

The Session GUI 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 a client that addresses it as `localhost`. This is 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.

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

3.4.4 Registration

The Session GUI is organized using a menu column with separate tabs for different groups of functions. One of the tabs is for communication with the Registrar, and in an unregistered individual it consists essentially of an item that can be clicked in order to register the individual in question.

Technically speaking, 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 entity-files 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.

Registration must be done from a session with the `remus` (kernel) agent. Once the individual has been registered, some minor changes need to be done in each of the other agents. These changes are done automatically during the start of the first session in each of those agents.

3.5 Leosocieties and 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*. Each attribute value is a KR-expression.

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. However, there is a tradeoff between this consideration and the need for freedom of naming in different applications and different user groups.

The notion of *leosocieties* is introduced in order to solve this problem. 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.

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

For communication between individuals in different leosocieties it may therefore be necessary to use a *translation service* for conversion between the name spaces of the respective leosocieties.

3.5.1 Network Description Entities

Although application-specific terminologies may be local to each leosociety, there is a need for standardization of those entities and relationships that pertain to the communication between individuals. Entities that are involved in the communication system itself include 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. In order to facilitate 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 conventions for network description entities are 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. `Novice-4`
- 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 are described in the documentation of network access.

3.5.2 Naming Conventions

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.

A natural 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.

The naming conventions for persons have been chosen according to those principles, and are as follows. An entity of the form

```
Lastname.Firstname
```

is used for characterizing generally known persons, for example

```
Einstein.Albert
```

If a disambiguity should arise then a distinguishing feature is added as a postfix, for example the person's year of birth. To be precise, 'Lastname' should be the family name and 'Firstname' should be the given name, regardless of the order in which these names are usually presented in the cultural context of the person in question.

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.

The namespace for network description entities is always maintained by the registrar of the leosociety in question. By way of exception, the namespace for individuals may also be maintained by a separate individual to which the registrar has delegated this task.

3.6 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. More about this will follow in the next chapter.

3.7 Describing Simple External Facilities

It was mentioned above that one of the installation steps is to "edit the two entityfiles in the Leohost directory." This refers to the files `facilities-catal` and `host-descr.` In both cases it should be clear from the default contents

of these files what changes may be needed. Additional details about the contents of the `host-descr` file are given in Section 4.4 in the next chapter.

The contents of the file `facilities-catal` are essential for the proper operation of the Session GUI. 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. A `facilities-catal` file in a Linux host will only make use of the values for `linux-commandphrase` and not the ones for `win-commandphrase,` and conversely for a Windows host. However, the standard distribution of the Leohost directory provides default values for both attributes in order to facilitate the task of adjusting them to the conditions in the host at hand.

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]
-------------------------------------------------------
```

Each individual stores this file in its `indiv-self-kb` knowledgeblock, which

means that it 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. Information of these kinds is normally entered during the original installation of an individual and using the Session GUI. If information about these entities 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.

Chapter 4

Additional Facilities for Self-Description

The present chapter will describe the information in the `Indiv-self` directories of Leonardo individuals and in the `Leohost` directories of computer hosts where Leonardo individuals operate, together with how this information is used by various services in the individual. Please recall that the `Indiv-self` directory is one of the top-level subdirectories within the individual, located sideways with the `Indivmap` directory and the directories for the various agents in the individual.

Please recall also that different individuals have in principle the same contents in their `Indivmap` directory, since it contains descriptions of the network where they are all located, whereas the contents of the `Indiv-self` directory are specific to each individual. The contents of the `Indivmap` directory are used for agent-to-agent communication which is documented in a separate report.

4.1 The `Indiv-self` and `Leohost` Directories

The `Indiv-Self` directory contains three essential subdirectories:

- `clonemon,` containing information for monitoring whether an individual has been copied so that it is a clone of an existing individual;
- `ownhist,` containing entityfiles with the history of major actions and events that have occurred in the individual;
- `revisions,` containing records of software revisions ie. software updates and changes of the overall structure of the individual.

In addition it contains the following entityfiles:

- `access-info,` containing the passwords for the individual and for services that it may wish to access;
- `ext-resources-catal,` containing URL:s and other access-path information for some of the services whose passwords are stored in the file `access-info`

- `facility-preferences,` which was described in Chapter 3, Section 3.8. It contains the user's preferences eg. for the choice of web browser and of text editor;
- `indiv-descr,` containing a number of attributes describing the individual itself, and some locally defined entities;
- `indivmap-amend,` which is used for system-internal purposes in the context of communication with the Registrar;
- `indiv-self-onto,` containing type declarations for the entities that are used in the various entityfiles of `Indiv-self` ;
- `masters-catal,` containing entities representing persons that may relate to the individual as its owner, as temporary users, or as owners of hosts that the individual has visited;
- `websites-info,` containing URL, username, and encrypted password information for websites as a service to the owner/user of the individual.

Finally there is the `indiv-self-kb` file containing catalog information about these other files, and the `cl` subdirectory which is used for the direct CommonLisp representation of the entityfiles, as usual.

The `Leohost` directory contains the following files:

- `host-descr,` containing the essential attributes describing the computer where the file resides;
- `facilities-catal,` which was described in Chapter 3, Section 3.8. It contains access paths to web browsers and other similar facilities that are available for use in the host at hand;
- `winleo-remus.cl,` which is used in Windows environments for invoking the remus agents of arbitrary individuals on Allegro CommonLisp by right-clicking the icon of the individual;
- `acleo-remus.cl,` which is used similarly for Allegro CommonLisp on Linux;
- `cleo-remus.cl,` which is used similarly for CLisp on Linux.

The additional files besides `facilities-catal` will be described in Section 4.4 below. There may also be other files whose names begin with `winleo-` , `acleo-` or `cleo-` and which are used for invoking other agents besides the `remus` agent. Further files may be added by various Leonardo-based applications.

The entityfiles in these directories are organized using the files `indiv-self-kb,` `indivmap-kb,` and `leohost-kb,` respectively. (The subdirectories of the `Indiv-self` directory use other arrangements, however). These are the files and subdirectories whose function will be described in the present chapter.

4.2 Self-Description in Leonardo Individuals

4.2.1 The `indiv-descr` file

The `indiv-descr` file contains at least two entities, namely the entity `indiv-descr` itself, and `current-individual`. The latter is the carrier of the individual's basic information about itself; the following is an example of this entity in an ordinary, *already registered* individual.

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

[: type self-individual]
[: in-leosociety soc.root]
[: in-individual (leoind: Novice soc.root 4)]
[: has-indiv-config Novice]
[: directory-name Novice-4]
[: created-on "2012-05-30"]
[: has-owner /Sandewall.Erik]
[: registration-state is-registered]
[: in-host host.dell-dc1]
-----------------------------------------------------
```

These *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, and it is used for identifying the individual in agent-to-agent communication. This is usually, but not necessarily the name of the file-system directory containing the individual.

The attribute name `in-individual` is ill-chosen and `is-individual` would have been better, but it is being used in a variety of places and therefore it has not been changed.

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: Novice soc.root 4)]
[: has-indiv-config Novice]
[: created-on "2012-09-14"]
[: registration-state is-individualized]
[: old-directory-name Novice-4]
[: 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.

4.2.2 Introduction of Network Describing Entities

When an individual is personalized it may be required to introduce entities for, for example, the host where the individual is currently located, and the owner of the individual. This information and these entities shall eventually be transmitted to the Registrar, but the system has been designed so that personalization and registration are separate operations, and so that it shall be possible for an individual to operate even before it has been registered. This means that there must be some way of storing network describing entities such as these locally until registration is performed.

This need is solved by allowing the personalization operation to add entities to the `indiv-descr` file that was described in the previous subsection. Then, during the registration operation, they are uploaded to the Registrar, re-obtained in the `Indivmap` files that are downloaded from the Registrar, and subsequently removed from the `indiv-descr` file. All of this happens automatically of course.

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 sometimes be resolved on this basis.

The Registrar also provides a directory name for the registered individual, for example `Novice-9` or `Cappa-12,` as discussed in the previous subsection.

4.2.3 The Individual's Own History

The `Indiv-self` directory in an individual has a subdirectory `ownhist` that is dedicated to preserving the history of major actions and events of all agents in the individual. Each session in any of the agents generates a historyfile whose name may be for example `ownhist-remus-184-0096,` for session number 89 in `remus` agent number 184. The registrar maintains a sequential numbering of all the kernel agents, regardless of the numbers of the individuals within their respective species.

These history files shall be considered as a resource that is available for use when an application requires it, but otherwise there is no need to worry

about them. The minimal contents of a historyfile is that it will contain four entities, such as

```
(ownhist: remus-184 96)
(session: remus-184 96)
(sysepisode: remus-184 96 0)
(sysepisode: remus-184 96 1)
```

All these entities have composite names, therefore. The first entity is the name of the entityfile, so it is the entity representing the filename `ownhist-remus-184-0096` and it has an attribute for the list of all entities in the file. The second entity represents the session as a whole, and specifies for example what was the host where the session executed. The remaining two entities represent *episodes,* viz. periods of time in the life of the agent in question. Each episode has a starting time, an ending time, and a sequence of action instances that were executed during the episode in question. In the default case that sequence is empty, but both the basic Leonardo software and various applications may register actions within the episode. However, the intention is that only major actions are to be registered, and not all the trivial actions during a session.

It is possible for an application to subdivide a session by ending the current episode and starting a new one. Episodes may also be nested hierarchically. In the minimal case, however, there is one episode labelled as number 0 for the time period between the end of the previous session and the beginning of the present one, and another episode, labelled number 1, for the duration of the present session. The inclusion of episode number 0 means that if an application is able to identify things that must have happened in the period preceding the current session, then it is able to add that information to the number 0 episode.

The starting and ending times of an episode are represented as sequences of numbers, for example

```
<2012 11 23 15 48 40 4 nil 7>
```

for the date of 2012-11-23, at 15.48.40 hours GMT, done in a timezone that is +7 hours relative to GMT, so that the local time was 22.48.40. The use of GMT as the primary time facilitates the comparison of timepoints that were observed in different timezones.

4.2.4 Revision Information

The `Indiv-self` directory in an individual also has a subdirectory `revisions` that is used for maintaining information about updates of the central parts of the individual, in particular its `remus` and `utilus` agents. We distinguish between *main revisions* and *subrevisions.* In a main revision, all contents in these two agents are removed except those parts that contain information that is specific to the agent, which is largely speaking the `Agent-self` subdirectory of the agent directory. The removed information is replaced by new definitions for everything. In a subrevision, on the other hand, only specific entityfiles are replaced or added.

The `revisions` directory contains information about the current revision and subrevision of the individual, and it is also used when a subrevision is received and applied. Each revision in the history of the Leonardo system is

identified by the date when it was issued. At the time of writing this report, the current revision is `revision-2012-11-22` so an entityfile with this name shall be found in the `revisions` directory. Files for earlier revisions may also remain there, but are only of historical interest. The file for the current revision contains one entity for the current revision, and then one entity for each of the subrevisions that have been imposed in the current individual, so the list of contents may be for example

```
(revision: 2012-11-22)
(subrevision: 2012-11-22 1)
(subrevision: 2012-11-22 2)
(subrevision: 2012-11-22 3)
```

The revision entity has an attribute for the current number of subrevisions that have been applied in the individual at hand. Each of the subrevision entities specifies what entityfiles are added or replaced for the purpose of that subrevision. The procedure for imposing a subrevision is therefore to replace the revision file with an extended one, containing an additional subrevision entity, and then doing the additions and replacements specified by the additional subrevision.

4.3 Self-Description of Agents

Major parts of the self-description are organized on the level of the individual, but there is also a need for some self-description for each agent within an individual. This is arranged in the `Agent-self` subdirectory of the directory for the individual. The most important entityfile in this respect is called `selfdescr` and it is located in, for example,

```
Leonardo/Novice-4/remus/Agent-self/main/Defblock/selfdescr.leo
```

The following is an example of the main entity in such a file.

```
-----------------------------------------------------
-- remus-184

[: type leo-agent]
[: in-leosociety soc.root]
[: in-individual (leoind: Novice soc.root 4)]
[: indivmap-name (kercl: soc.root 1 soc.root 184)]
[: leoname remus]
[: primary-language Swedish]
[: is-server-in <dres.sesgui>]
[: self-location "../../../remus/"]
[: leoprovider "../../../remus/"]
[: created-on "2012-11-14"]
-----------------------------------------------------
```

The number of this entity, such as 184 in the example, indicates that this is number 184 among all the individuals that have been registered for the present leosociety. The contents of the `selfdescr` file are mostly for technical purposes in the operation of the agent and the individual.

4.4 Hosts and Devices

4.4.1 The Host Description in the `Leohost` Directory

The major entityfile in the `Leohost` directory is called `host-descr` and the following is an example of the two entities contained there.

```
------------------------------------------------------------
-- current-host
[: type host-self-type]
[: host-own-name alpha]
[: has-host-type detachable-host]
[: has-owner-string "Sandewall.Erik"]
[: has-allegro-path "C:\acl81\"]
------------------------------------------------------------
-- (host-in-soc: soc.root)
[: type host-self-type]
[: host-own-name host.alpha]
[: has-owner /Sandewall.Erik]
------------------------------------------------------------
```

These attributes are entered during the personalization operation, and can be updated using the Session GUI. The purpose of the second entity is to accomodate situations where the same host is used in several leosocieties, in which case there shall be one `(host-in-soc: .)` entity for each of them. This makes it possible for the same host to have different names in different leosocieties, and similarly for the identifier for the owner.

When a session with a Leonardo individual is started in 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 of the society-specific identifier, 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, 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.

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 pos-

sible 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.

4.4.2 Description of Detachable Memory Devices

Detachable memory devices, including both USB sticks and external hard disks, can be assigned names in current operating systems. From the point of view of a Leonardo individual, the default name for such a device has the form `dmd.devicename` where `devicename` is the name assigned by the operating system, but always converted to lower-case letters.

Under the Linux operating system it is straightforward for an individual to identify the default devicename, since detachable devices are found under `/media/devicename/` in the directory structure. However, device names are not so readily available under Windows systems, which has led to the following conventions for the description of detachable memory devices (dmd).

The top directory level of a dmd that is used by Leonardo should contain two specific files. First, one file that is always called `medianame.txt` and that merely contains one line of text, namely, the name of the dmd, in lower-case letters, but without the `dmd.` prefix. Secondly, an entityfile containing a simple description of the dmd in question. The name of this file shall be based on the device name, so that if the device name is `kingston-6,` for example, then the name of the description file shall be `kingston-6-descr,` with the `.leo` extension as always.

The following is an example of the contents of such a dmd description file:

```
------------------------------------------------------------
-- (dmdfile: kingston-6)

[: type dmd-descr-file]
[: contents <(dmdfile: kingston-6)>]
[: has-leonardo-dirs <Leonardo Leonardo-copies Leonardo-old>]
[: owned-by /Sandewall.Erik]
------------------------------------------------------------
```

Notice therefore that it is the composite entity `(dmdfile: kingston-6)` that represents this entityfile, and that the name `kingston-6-descr` is just the filename and nothing else.

The `has-leonardo-dirs` attribute shall contain a set (or sequence) of the directory paths for those directories in the device that may contain Leonardo individuals. This is so that a routine that identifies Leonardo individuals in this device can limit its search to the relevant directories.

The command `namd` can be used for generating these files when working in a Windows context; it is used as in the following example

```
namd E kingston-6
```

Its effect is to set up the required information in the dmd currently identified as volume `E:/` except that the value or `has-leonardo-dirs` must be assigned manually by the user.

4.4.3 Remote Invocation

Although the basic way of starting a session with a Leonardo individual is to invoke one of the invocation files in the `Agent-self/main` directory, this method is also a bit clumsy. A more convenient way is to right-click the icon for the individual in question and to select the desired agent in this individual.

Two things are required in order to be able to start sessions in this way. First of all, an appropriate *startup file* must be located in the `Leohost` directory. Therefore, one must first await that this directory is in place.

One such file is `winleo-remus.cl` which is provided in the standard distribution and has the following contents.

```
(prog (rd agent)(setq agent "/remus/")
      (setq a (system:command-line-arguments))
       ;; returns eg. ("C:\\acl81\\mlisp.exe" "C:\\Leonardo\\Novice-4")
      (setf (get 'invoc-ent 'indiv-dir)
            (concatenate 'string (cadr a) "/") )
      (setf (get 'invoc-ent 'agent-dir)
            (concatenate 'string (cadr a) agent) )
      (load (concatenate 'string (cadr a)
                                 "/remus/Startup/cl/winleo.leos" ))
      nil )
```

The example shows that the contents of this file are specific to the use of `winleo` startup and for the `remus` agent, but it is not specific to a particular individual; the choice of individual is obtained from the command-line arguments.

In order to use this file, one should invoke the CommonLisp system in question with a shell command like the following one:

```
C:\acl81\mlisp.exe -L C:\Leohost\winleo-remus.cl -- C:\Leonardo\Novice-4
```

This can be done in either of two ways. One way is to define a `.bat` file with exactly these contents. This will obtain an invocation file that just starts the `remus` of `Novice-4,` and nothing else. The other way is to define an option for right-clicking the icon of an arbitrary individual, and not merely `Novice-4,` and get it to produce and execute a shell command like the one shown above. Still in the case of Windows, to arrive to the place where such options are defined, one should type `shell:sendto` into a directory browsing window, thereby obtaining a set of icons representing the choices when you right-click a directory icon and select the `sendto` option there.

In this place one shall create one more shortcut, name it appropriately, right-click it and select its `Properties` aspect, go to the `Shortcut` tab of the resulting `Properties` card, and put the following line into the `Target` field there:

```
C:\acl81\mlisp.exe -L C:\Leohost\winleo-remus.cl --
```

Elsewhere under `Shortcut,` the `Start in` field shall be empty, the `Shortcut key` shall say `None,` and the `Run` field should select `Normal window.`

Once this is in place, things are ready for invoking agents by right-clicking the icon of the individual. Scripts for other agents and other invocation files

can easily be done in similar ways, and the approach is also easily extended to the Linux environment and the use of CLisp.

The use of startup files in the `Leohost` directory may also be useful, in particular while working under Linux, since some Linux versions do not allow a session to be started from a directory in a detachable memory device. This problem can be circumvented by using a startup file in the `Leohost` directory, as shown above.

The effect of the code stub in files such as `winleo-remus.cl` is to define the baseline relative to which entityfiles in the Leonardo individual shall be read and written. This overrides the baseline that is obtained by default when a session is started using an invocation file in the `Agent-self` directory.

4.5 Session Preferences and Configuration

The representation of user preferences for the choice of external facilities has already been described in Chapter 3, Section 3.8. The following are some additional ways of stating user preferences for particular aspects of a session with a Leonardo individual.

4.5.1 Communication Language

The Leonardo system is in principle designed so that it shall be possible to use other languages, besides English, both in the Command-Line Executive and in the Session GUI. This requires that all messages from the system are defined in each of the languages being supported, and not merely in English. The machinery for achieving this exists, but the work on restating existing phrases in additional languages has only just started. This is therefore not yet a practically useable feature, but it is possible to try it out.

The primary command language for an agent is set in its `selfdescr` file, as seen from the example in Section 4.3. The command for changing language in the midst of a session is like in

```
setlang English
```

The available alternatives at present are `Swedish` and `French,` besides of course `English.` A small number of phrases have been defined in Swedish, and a very small number in French, just for getting started.

Phrase definitions may be found either in the respective program files where the phrases are used, or in entityfiles in the `Startup` directory with names such as `corephrases-swedish` and `corephrases-french.`

4.5.2 Session Mode Settings

The file at the following location

```
Leonardo/.../remus/Agent-self/main/State/cl/sessionmode.cl
```

for an arbitrary individual contains settings for a number of global Lisp variables that affect the general operation of the system and the session. Several of these are settings that Leonardo does for the hosting CommonLisp

system and should not be changed, but a few of them may be of interest for an end user. They are as follows, with their standard value assignments.

```
(setq *block-concurrent-sessions* t)
(setq *trace* nil)
(setq *no-leoprint-babble* t)
(setq *no-startup-babble* t)
(setq *base-server-port* 8080)
```

Presently the `sessionmode` definition in the `remus` agent applies for all the agents in the individual. The effects of changing these values are as follows.

- `*block-concurrent-sessions*` - if set to `nil` then concurrent sessions with the same agent are admitted. This is sometimes very useful when debugging changes in the most basic parts of the Leonardo software.
- `*trace*` - if set to `t` then trace messages are produced in a variety of places in the system, and in particular in the routines for agent-to-agent communication, as a means of facilitating the debugging of such communication.
- `*no-leoprint-babble*` - if set to `nil` then trace messages are generated when entityfiles are written.
- `*no-startup-babble*` - similarly for trace messages during startup.
- `*base-server-port*` - the port number for the server for the Session GUI is selected as the number indicated here, and subsequent numbers if several agents in the same host are running concurrently and opening their Session GUI.

4.5.3 The Startup Configuration Script

When a session with a Leonardo agent is started, using either an invocation file or by right-clicking the icon of the individual, the standard routine is that important basic entityfiles are loaded, the Command-Line Executive is started, and then it is up to the user to load whatever other knowledgeblocks and programs he or she wishes to use. However, for some applications it may be desirable to define extensions to the startup sequence, so that additional knowledgeblocks are loaded automatically, or other actions are taken before the CLE starts.

This can be done and is particularly useful in higher-level agents. To understand how it is done, the reader should first inspect the `Startup` directories of the `remus` agent and the `utilus` agent of a Leonardo individual. Notice the following:

1. The `remus/Startup` directory contains entityfiles with names such as `winleo.leo` and `acleo.leo` which are names that correspond to the invocation files that are found in the `remus/Agent-self/main` directory. These are called *invocation entity files.*

2. The `utilus/Startup` directory does not contain any such invocation entity files.

3. However, the entityfile `startup-catal` in `utilus/Startup` does contain definitions of the locations of the invocation entity files, but it refers to their locations in the `remus/Startup` directory.

This is explained as follows. Each agent has its own set of invocation files, and it is able to generate them from the invocation entityfiles that it uses, as defined by its own `startup-catal` file. Therefore, several agents may share the same startup entityfiles, but not the same invocation files.

Consequently, in order to define an extended startup sequence for an agent, such as the `utilus` agent, one must first of all arrange that it obtains its own, separate invocation entity files. This is done by copying eg. `winleo.leo` from `remus/Startup` to `utilus/Startup` and changing the location specification in the `startup-catal` file of `utilus` so that it will contain, in the example,

```
------------------------------------------------------------
-- (location: winleo)
[: type location]
[: in-directory "../../../utilus/Startup/"]
[: has-filename "winleo"]
[: filename "../../../utilus/Startup/winleo"]
------------------------------------------------------------
```

After this, one must do, in a session with the `utilus` agent,

```
    updfil startup-catal
    updfil winleo
```

This will regenerate the underlying `.cl` files and the invocation file `winleo.bat.` After these operations, one can use the `winleo` invocation file just like before, and the only change is that now it will load `utilus/Startup/cl/winleo.leos` instead of `remus/Startup/cl/winleo.leos` at the beginning of the session startup sequence.

This makes it possible to configure the startup as desired. It is done by adding a property like the following in the `winleo.leo` file of `utilus,` just before the line of `oooooo` characters that marks the end of the file.

```
@Startscript
[soact
    [stagui]
    [loadk reasoning-kb]
      ]
```

The property defined here shall be an action expression, as briefly described in Chapter 1 and more in detail in the separate report on script definitions in Leonardo. The operator `soact` that is used in the example means *sequence of actions,* and its 'arguments' are action expressions. These are the same kinds of action expressions as are input to the Command-Line Executive, except that they must be delimited by square brackets. Unlike the CLE input they may also run over several lines.

When the `winleo.leo` file has been modified in this way it is necessary to again do

```
    updfil winleo
```

After this, the next time a session is started e.g. by invoking `winleo.bat`, the startscript will be executed at the end of the startup process, meaning that the Session GUI will be started and the simple reasoning package `reasoning-kb` will be loaded.

The same kinds of extensions may of course be done in other agents besides `utilus`. The Startscript is specific to each agent, so that although a `utilus` agent can load all the knowledgeblocks in the underlying `remus` agent, a Startscript that is defined in a `remus` agent will only apply to sessions with that agent and with other agents that use its invocation entity files, and not to sessions in agents that use other invocation entity files.

If agents with startup scripts are to be invoked using the right-click option that was described in Subsection 4.4.3 then one must observe that the change of invocation entityfile, such as `winleo.leo`, will affect the contents of the last line in the remote invocation script. In the example above it would have to be changed to

```
(load (concatenate 'string (cadr a)
                           "/utilus/Startup/cl/winleo.leos" ))
```

provided that `utilus` is the agent for which the startup script has been defined.

4.6 External and Internal Facilities

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, in particular in the `utilus` agent, and they may also be downloaded from the Registrar or from other sources. The usual organization of an individual is that it contains at least three layers of agents: the `remus` agent which is the lowest one, the `utilus` agent that comes next, and then all other agents which may or may not be located on top of each other, but they should all be 'above' the `utilus` agent. Higher level agents may load knowledgeblocks in agents that are below it.

Given this hierarchy, the standard practice is to keep the `remus` agent as fixed as possible, and to consider the `utilus` agent as a kind of module library, so that application agents can use the knowledgeblocks that are in it. The standard distribution contains already a number of such library modules in the `utilus` agent, and there is also a facility for downloading further knowledgeblocks and placing them in the `utilus` agent. Sessions with the `utilus` agent are therefore mostly used in order to request the download of knowledgeblocks to its library.

Some simple external facilities were described in the previous chapter. More

advanced external facilities as well as internal facilities are described in a separate memo, *Facility Installation in Leonardo.* External facilities are provided using installation scripts that perform some or all of the operations that are needed for obtaining the external facility.

4.7 External Resources

Whereas both external and internal *facilities* 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.

4.7.1 Description of External Resources

For each external resource there must be a description with respect to the URL where the resource can be accessed, the format of the requests that can be sent to it, and the format or procedure for decoding the responses to those requests. The rules for this are sometimes quite simple and sometimes complicated, and in many cases the exact representation of this has to be left to the knowledgeblock or other module that provides the interface to the external resource. However, in those cases where it is desirable to place such access-path or access-procedure information in a generally used location, two possibilities are offered. The entityfile `foreign-servers-catal` in `indivmap-kb` is recommended if the server in question is of general interest for the leosociety at hand, and the entityfile `ext-resources-catal` in `indiv-self-kb` is appropriate if the server in question is of general use in the current individual but not beyond.

4.7.2 Password Management

Many external resources 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 updating the information about their usernames and passwords.

4.7.3 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. The user's password information is stored in the entityfile `websites-info` in `indiv-self-kb.`

4.7.4 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. The internal facility for E-mail is described in the separate report on agent-to-agent communication.

4.8 Working with Leonardo Configuration Information

The present chapter has described how the configuration information in Leonardo is organized into entityfiles that the user can inspect and modify freely. Some of this information can be accessed using the Session GUI, but in other cases on must edit the self-description entityfiles directly in order to configure the system to one's own preferences. The following are some practical tips in this respect.

The standard procedure for doing this is simply to text-edit the entityfile and then to load it into one's session using the `loadfil` command that was described in Chapter 1. However, in some cases it is necessary to also let this `loadfil` command be followed by a `writefil` command for the same file. This has the effect of writing the file back, which may seem redundant, but the point is that this will also write the corresponding `.leos` file. These files were described in the last section of Chapter 1. The reason why this is important is that some entityfiles are initially loaded in their `.leos` manifestation, namely, those files that are loaded so early during the session startup process that the parser for KRE expressions has not yet been loaded. For those files it is essential that the `.leos` file is also updated, which is what the `writefil` command achieves.

In addition, the `writefil` operation on an invocation *entityfile* such as `winleo.leo` has the side-effect of generating the corresponding invocation *file* such as `winleo.bat` in the agent where the `writefil` operation is performed. Please recall that this is not necessarily the same agent as where the invocation entityfile is physically stored, since several agents may share the same invocation entityfile but each of them obtains its own invocation file.

Entities in Leonardo may be simple or composite, and this applies for entities that represent entityfiles as well. The present chapter has mentioned some cases where entityfiles have composite names, for example history files which are identified by entities such as `(ownhist: remus-184 96)` . The *filename* of such an entityfile as used in the `ownhist` directory will be `ownhist-remus-184-0096,` but this is only in the file directory and it is not a Leonardo entity. Commands such as the following one

```
loadfil ownhist-remus-184-0096
```

will not work therefore, and one has to do instead

```
loadfil (ownhist: remus-184 96)
```

Composite entities tend to be lengthy, and if the same entity is used repeatedly then it may be convenient to use session variables, for example

```
ssv .hr (ownhist: remus-184 96)
loadfil .hr
writefil .hr
```