CASL

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

Erik Sandewall

The Leonardo Kernel and Platform

This project memo pertains to the development of the Leonardo system. Identified as PM-leonardo-005, it is disseminated through the CAISOR website and has URL `http://www.ida.liu.se/ext/caisor/pm-archive/leonardo/005/`

Related information can be obtained via the following www sites:

CAISOR website:	`http://www.ida.liu.se/ext/caisor/`
CASL website:	`http://www.ida.liu.se/ext/casl/`
Leonardo system infosite:	`http://www.ida.liu.se/ext/leonardo/`
The author:	`http://www.ida.liu.se/~erisa/`
Date of manuscript:	2008-10-06

1 Introduction

This report describes the functioning of the Leonardo *kernel* and *platform*, that is, those parts of the system that are not specific to one of the major applications. The following earlier report is assumed to be previously read:

- Introduction to the Leonardo System and Overview of Major Applications

The presentation in the present report describes the system on the level of the LDX notation and does not dig down into the level of the host programming language (which is CommonLisp). The following report is based on the present one but does address host-language issues; it is therefore recommended as continued reading:

- The CommonLisp Implementation of Leonardo

Most of the contents of this report have been taken from the second half of an earlier report (reference) that was written in 2006. The description of the platform is therefore incomplete since it has developed significantly since then.

2 The Leonardo Kernel

Requirements according to the Design

The following were and are *the requirements on the kernel*:

- It shall contain self-describing information and corresponding procedural capabilities whereby it is able to administrate itself, its own structure, and its own updates.
- It shall provide the extension capabilities that make it possible to attach additional knowledge blocks to it and to administrate them in the same way as the kernel administrates itself.
- It shall provide adequate representations for the persistent storage of all contents of the blocks in the kernel, as well as the representations and the computational services for performing computations on the same contents.
- It shall provide capabilities for adaptation, in particular to facilitate moving a system between hosts, and for defining alternative configurations based on different sets of knowledge blocks.
- Although the experimental system will be based on an existing, conventional operating system and ditto programming language, it shall be designed in such a way that it can be ported to the weakest possible, underlying software base.

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

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

The Structure and Constituents

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

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

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

The Core Knowledgeblock, `core-kb`

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

- The initial loading or 'bootstrap' machinery. It consists of a few entityfiles that are the very first ones to be loaded when an activation is started, and it prepares the ground for subsequent loading.
- The index file of the core knowledgeblock. (`core-kb`).
- The ontology file of the core knowledgeblock, which is at the same time the core or "top-level" ontology of Leonardo as a whole. (`coreonto`).
- Miscellaneous additions to the core ontology that are needed for historical or other reasons. (`toponto`).
- Definitions of procedures for loading entityfiles and parsing the textual representation of entities. (`leo-preload, leoparse`).
- Definitions of elementary operations on the structured objects in the Leonardo data representation, such as sequences, sets, and records. (`leoper`).
- Miscellaneous auxiliary functions that are needed in the other entityfiles but which have a general-purpose character. (`misc`).

- Major timepoints in the history of the present instance of the system (`mp-catal`).
- Functions for administrating entities, entityfiles, and knowledgeblocks, for example, for creating them and for editing their attributes. (`leo-admin`).
- Functions for writing the textual representation of entityfiles, and for producing the textual representation of Leonardo datastructures from their internal ones. (`leoprint`).
- Definitions for a simple executive for command-line operation of the system. (`lite-exec`).

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

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

The Leonardo Startup Machinery

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

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

The first three of these aspects is implemented using the `core-kb` knowledgeblock; the fourth one using the separate `syshist-kb` knowledgeblock.

Notice, however, that the first aspect is a step towards (i.e., facilitates greatly) the fourth one.

The startup process for Leonardo activations is actually a good illustration of how a somewhat complex process can be organized around its symbolic data structures. Appendix 1 describes this in some detail.

Configuration Management

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

Each configuration may also make some other specifications, for example for extra information that is to be loaded in order to start it. Furthermore, each configuration shall specify its user interface, in the sense of a command-line interpreter, a GUI, and/or a web-accessible service. This is done with the `execdef` attribute on the startup-file that was described in Appendix 1.

The Knowledgebase Index Files

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

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

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

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

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

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

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

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

It defines the location of the entityfile `leoadmin` by introducing a composite entity `(location: leoadmin)` whose type is `location`, and assigning a `filepath` attribute to it([1]). Among the files that occur at the beginning of the startup phase, `self-kb`, `kb-catal` and `core-kb` consist mostly or entirely of such entities.

3 Other Kernel Knowledgeblocks

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

Both reproduction and version management are essential for the evolution of the Leonardo software through concurrent strands of incremental change in several instances of the system, i.e. several Leonardo individuals. This is the decisive factor for considering these to be an integral part of the system kernel. In addition, by doing so we also provide a set of tools that can be used in applications of several kinds. – The importance of having software tools for version administration do not need to be explained; it has been proven through the very widespread use of tools such as CVS ([2]).

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

Awareness of Time in the Leonardo Individual

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

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

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

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

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

System History and Version Management

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

Local version management works as follows. The individual maintains a sequence of *archive-points* which are effectively a subset of the timepoints that are registered by `chronos-kb`. Archive-points have names of the form `ap-1234`, allowing up to 9999 archivepoints in one individual. Each archive-point is associated with the archiving of a selection of files from one particular knowledgeblock. The archiving *action* takes a knowledgeblock as argument, obtains a new archivepoint, and for each entityfile in the knowledgeblock it compares the current contents of the file with those of the latest archived version of the same file. It then allocates a new directory, named after the new archive-point, and places copies there of all entityfiles where a nontrivial difference has been identified. The archive-point is an entity that is provided with attributes specifying its timepoint, its knowledgeblock, the set of names for all entityfiles in the knowledgeblock at the present time, and the set of names for those entityfiles that have been archived.

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

Local version management is useful for backup if mistaken edits have destroyed existing code, but it does not help if several users make concurrent changes in a set of entityfiles. This is what *two-party version management* is for. In this case, there is one 'server' individual that keeps track of updates by several users, and one 'client' that does its own updates and sometimes 'synchronizes'([3]) with the server. Such synchronization must always be preceded by a local archiving action in the client. Then, downward synchronization allows the client to update its entityfiles with those changes

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

that have been incorporated into the server at a time that succeeds the latest synchronized update in the client. If the current entityfile version in the client is not a direct or indirect predecessor of the version that is presently in the server, then no change is made. After that, an upward synchronization identifies those entityfiles whose contents still differ between the server and the client. If the version in the server precedes, directly or indirectly, the current version in the client, then the current version in the client is imposed on the server.

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

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

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

Configuration and Reproduction of Individuals

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

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

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

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

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

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

For offspring, the following conventions are adopted. The making of an offspring from an individual is to be considered as an action of that individual, and is to be recorded in its history. Each individual has a *name*, and the offspring of a particular individual are numbered from 1 and up. No individual is allowed to have more than 999 offspring. The first individual under this scheme was called `lar`, and its direct offspring are called `lar-001`, `lar-002`, etc. The offspring of `lar-002` are called `lar-002-001`, `lar-002-002`, and so forth. The abbreviation `lar` stands for "Leonardo Ancestry Root".

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

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

In principle, each new individual should obtain a copy of all the software of its parent. In practice this is quite inconvenient when several individuals are stored on the same host; one would like them to be able to share some of the software files. This has been implemented as follows: Each individual may identify another individual that is known as its "provider", and when its index files specify the locations of entityfiles, they may refer both to files in its own structure, and files in its provider. An individual is only allowed to update entityfiles of its own, and is not supposed to update entityfiles in its provider ([4]). When a new individual is created, then it is first produced

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

with a minimal number of files of its own, and it relies on its parent as its provider for most of the entityfiles. After that, it is up to the offspring to copy whatever software it needs from its provider to itself, until it can cut that umbillical cord. Only then is it in a position to migrate to other hosts. Besides, given adequate software, it may be able to import knowledgeblocks and entityfiles from other individuals and not only from its parent.

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

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

Other Facilities in the Kernel

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

4 The Leonardo Platform

The next layer in the Leonardo software architecture, after the kernel, is called the *platform.* This layer is under construction and is intended to be open-ended, so that new contributions can be added continuously as the need is identified and the implementation is completed. The following are some platform-level knowledgeblocks that existed and were used in late 2006 (developments since then are not yet documented).

Channels

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

risk.

Communicable Executive

One of the examples in the report *Introduction to the Leonardo System and Overview of Major Applications* concerns communicating and cooperating Leonardo individuals. This example was done using the *communicable executive* (CX). The basic command-line executive in the kernel is not sufficient for it. CX performs incessantly a cycle where it does three things:

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

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

Appendix 1: Startup of Activations

Problem and Approach

It is particularly attractive to implement Leonardo in interpretive and 'script'-type programming languaes, such as CommonLisp and Python, but then the implementation of the startup machinery along the lines that were described in section 2 of this report, involves an interesting chicken-and-egg problem. In principle, you want the interpreter of the programming language to be able to read, parse, and digest textual entityfiles, and if an entityfile contains e.g. function definitions then those definitions shall take effect along the way when the file is loaded. More generally, if an entityfile contains an executable expression in the programming language, then that expression is to be executed, since typically the way to define a function is to execute an expression that stores the definition.

Consider now the very first entityfile that is going to be loaded into the interpreter at hand (CommonLisp, Python, etc) in order to start the bootstrap process. The interpreter has been invoked with that file as its argument, and that file is going to add other entityfiles that are needed. If this file is edited and then loaded into the Leonardo system, for example for program analysis or for version management, then the bootstrap process will start again. This is of course not intended.

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

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

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

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

Details of the Startup Process

The startup or 'bootstrap' process is actually a nice illustrative example of the declarative style of system design in Leonardo, and we shall therefore describe it in some detail.

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

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

In this sequence, `core-kb` is the first file for loading the core knowledgeblock and from there on, the system reads the entityfiles that are defined by the knowledgeblocks that were listed on pages 2-3. They are loaded in exactly the same way as for all other knowledgeblocks. The five simple files preceding `core-kb` are the only ones that have a special role. Going backwards, `kb-catal` is a catalog containing the physical locations (directory and filename) for the index files of all the knowledgeblocks in the individual at hand. This is needed so that the system can look up `core-kb` in order to load it, as well as for loading any other knowledgeblock. Before it, `self-kb` is a catalog of those entityfiles whose contents are specific to the present individual and/or host, in particular, the location of `kb-catal`. Again before, `selfdescr` is a small entityfile containing the name and ancestry of the individual at hand, so that it 'knows' what it is called and from where it was generated. The locations of `selfdescr` and `self-kb` are hardwired into the software.

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

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

Data-driven initial startup

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

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

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

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

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

or, in translation to conventional programming-language syntax:

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

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

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

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

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