CASL

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

Leonardo

Erik Sandewall

Self-Description and Self-Modification in the Leonardo Computation System

1 Introduction and Overview

The capability of a software system to describe its own structure and its own abilities is a necessity for the implementation of general-purpose intelligence in that system. However, on a more practical level it is also an important resource that makes it possible to implement a number of useful services for the user of the system, including configuration for specific user needs, version management, the installation of updates, and debugging aid.

We believe that a systematic and well thought-out design for self-description can serve both of these purposes: it can help to increase the quality of the system's services, and it can also constitute a platform for the eventual implementation of intelligent autonomy in the system.

The present report describes the results of an explorative effort towards a system that is self-describing on the practical level but in a comprehensive and systematic way, with a clean design that is as expressive as possible. We shall describe the self-description and configuration facilities in the *Leonardo system* [[1]] which is an executive environment that contains a number of software services, and where additional such services can easily be added. The purpose of the Leonardo system is:

- To provide a unified and convenient framework for data that is shared between services, or that provide configuration information for them.
- To provide a high-level scripting mechanism that makes it possible to automate the routine use of participating services.
- To make it possible to include Artificial Intelligence techniques for knowledge representation and for the dynamic execution of scripts, for example through automatic and case-based planning.
- To provide an "intelligent agent" framework where several Leonardo environments can communicate by message-passing in order to cooperate for the execution of given tasks.

The Leonardo system is also an experiment with a novel approach to the architecture of software systems. This aspect is described in some detail in another report (reference) and will not be addressed here.

Each operational instance of the Leonardo system is called an *individual.* It has two manifestations:

- A *persistent manifestation* as a directory structure (a directory and its subdirectories on all levels) containing files that represent data and "programs" that are used by the individual.
- A *dynamic manifestation* which is also called a *session*, i.e., a computational process that uses the information in the persistent manifestation and that is also able to modify it.

Normally there should not be more than one session of the same individual, that is, at most one session that is started from and uses a particular persistent manifestation at any one time. (Exceptions are made in low-level

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

debugging situations). An individual that does not have a session at a particular time is said to be *asleep* then. When we speak about message-passing between individuals in this report, we mean message-passing between sessions of those individuals. Message-passing to a sleeping individual may be implemented using a mailbox facility, but will not be addressed here.

One of the services that are included in a Leonardo individual is an http server. Conseqently, a Leonardo individual can be accessed in the following ways:

- Using a command-line interface.
- Using a web browser.
- By message-sending between individuals. Such message-passing has been implemented using the http protocol.

For example, it is possible to run several Leonardo individuals on the same computer, in which case they should use different ports for their http servers. Each of the individuals can be controlled directly using its command-line window. For situations where a graphical interface is preferred, it is recommended to implement it using the built-in web server and to access it using a browser running in localhost mode. Message-sending between individuals on the same computer is straightforward and can be interleaved arbitrarily with command-line and web browser interactions. Remotely located agents can also be accessed in all three modes: graphical mode and message-passing mode using their web servers, and command-line mode using an ssh connection.

Regardless of the mode of access, the user can make use of the same set of commands and scripts (commands defined in terms of other, more elementary commands). There is a *scripting language* called the *Generic Scripting Language* (GSL) that is used for this purpose. GSL commands can be typed into the command-line interface; the dynamic response to web-server requests from a browser can be defined using GSL, and request messages that are sent from one individual to another are also GSL commands.

Furthermore, the Generic Scripting Language is used as a basis on top of which a number of function-specific scripting languages are defined. For example, the Document Scripting Language (DSL) is based on GSL and is used for defining the textual contents of static and dynamic web pages, and also as a source language for research articles and reports such as this one. DSL is used for generating HTML, LaTeX, or other existing languages as required by the application.

As another example, the Operational Scripting Language (OSL) is intended to capture [2] a good part of the functionality of the shell script language of an operating system. The use of a single framework for scripting languages for several domains has advantages in terms of economy of implementation, ease of invoking scripts in one language from another one, and access to a common data and knowledge base. In a longer perspective it also suggests the possibility of using high-level techniques such as automatic composition of scripts to achieve given tasks (often called action planning) across the reportoire of different, function-specific extensions of the Generic Scripting Language.

[2]The design and implementation of this language is work in progress.

The scripting languages are not intended to replace the use of a conventional programming language, however, and they are not designed to have all the functionalities of a programming language. Instead, verbs in the scripting languages (where a command consists of a verb and its arguments and parameters) can be defined in several different ways:

- As a piece of program in the *host programming language* in which the Leonardo system has been implemented
- As a *script* in the scripting language at hand, in which case it is defined in terms of other verbs in that scripting language or another one
- Using a *dynamic scripting facility* that composes a script for the task at hand and proceeds to execute it immediately.

Each scripting language can therefore be restricted to those facilities that are best handled in the context of the other verbs and the data-structure facilities of that scripting language.

Several services have been implemented based on the Leonardo platform, including the following:

- The MADMAN system for preparation and management of articles, reports, and static webpages using the command-line interface and for use by a single user
- The KEPS system that provides a subset of those services but for a research group or other similar group of users, and using a webpage interface
- The (unnamed) system for managing the Common Knowledge Library (CKL), a web-based library of knowledgebase modules
- The CAPPA demonstration environment for the workflow between research author, institutional repository operator, journal publisher, and others
- The Parallell Publication Advisor, a website that provides advise about how the detailed requirements of a journal publisher concerning self-archiving by the authors are to be realized in practice, down to the level of generating a frontpage for the article that conforms to the publisher's requirements

2 Representation Framework

The representation framework contains the framework for representing information in a spectrum from simple data to information with more complex structure, as well as the framework for representing scripts (program-like structures typically consisting of a sequence of commands) that pertain to different system functions or application domains. In this section we shall briefly explain those aspects of the representation framework that are used for expressing self-describing information and self-modification operations. A more detailed description of the representation framework can be found in the lecture notes and reports that are available on the website for the Knowledge Representation Framework, KRF [3].

2.1 Knowledge Representation Expressions

A system with the characteristics of Leonardo needs a textual representation of structured data (also called a serialization of its data structures), besides the scripting language(s). In Leonardo the data representation is considered as the primary one, and the scripting languages are expressed using the syntactic constructs of the data language, with very minor extensions. In this way it is possible to use the same parser for programs and data, and scripts obtain immediately an internal representation as data structures that can easily be interpreted and manipulated.

Leonardo differs in one important respect from other languages that have already used this approach, such as Lisp, Scheme, and Prolog. Those languages use one single kind of bracket symbols, such as the round parentheses in Lisp. Leonardo uses several kinds of brackets with specific meanings. This gives us a much more readable notation for both data and scripts. From an abstract point of view this difference is trivial, since one can always encode all kinds of constructs using just one kind of brackets, but from the point of view of engineering a good notation, it makes a considerable difference.

Expressions in the data language for Leonardo are called Knowledge Representation Expressions (KRE) which is essentially an extended set theory notation. Expressions in KRE are constructed from symbols, strings and numbers which are composed recursively as in the following examples

```
<a b c>
{a b c d}
(op: a b c)
(fn a b c)
[op a b :tag1 v1 :tag2 v2]
[: a v]
```

These constructs are referred to as a sequence, a set, a composite entity, a form, a record, and a maplet, respectively. A mapping is represented as a set of maplets. The difference between composite entities and forms is that a form is evaluated in the usual, recursive manner by evaluating its arguments and then applying a function that is associated with the function symbol leading the form. A composite entity, on the other hand, is in itself an entity in the information structure at hand, and as such it can be assigned attributes consisting of a tag and a value. Composite entities are therefore used like terms in Prolog, for example; their operators can be thought of as Herbrand functions.

2.2 Information States, Entity Descriptions and Entityfiles

The operational datastructure in a Leonardo session consists of a set of entities and, for each entity, an assignment of *values* for each of a number of *attributes* and a number of *properties*. This entire structure at a given point

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

in time is called an *information state.* Attribute values are Knowledge Representation Expressions; property values are strings that typically consist of several lines. A property may be used e.g. for representing a program module in the host programming language, or for the documentation of that module.

An aggregate consisting of an entity, its attributes and properties and their corresponding values is called an *entity description.* An information state is therefore a set of entity descriptions. Within the session the information state is considered as one single structure without any particular partitioning, although there are some attributes that serve the role of catalogues and whose value shall be a KRE set of entities that have been grouped together. On the other hand, the persistent representation as text files does use a partitioning of the entity descriptions into *entity files* each consisting of a limited number of entity descriptions. The following is a simple example of an entity description in its textual representation:

```
-----------------------------------------------------------
-- yellow

[: type color]
[: examples {sun sunflower butter (juice-of: orange) gold}]
[: translations {[: german gelb][: french jaune]}]

@Comment
Pure gold is colorless but when gold is used in practice it
is usually in a alloy which gives it a warm yellow color.

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

This is an entity-description for the entity `yellow` which is assigned values for the two attributes `examples` and `translations`, besides the `type` attribute which must always be present. The same entity is also assigned a value for the property `Comment` which is a string consisting of two lines as shown in the example.

An entityfile can be understood abstractly as a sequence of entity descriptions, but its persistent manifestation in the computer is a textfile consisting of a number of entity descriptions of the kind just shown, separated by a line of dashes. Notice that this line of dashes is syntactically significant as a separator between entity descriptions; it is not just a kind of comment.

It is true that one could have used a more systematic notation where the entire entityfile is expressed as a KRE mapping from entities to mappings expressing their value assignments for attributes and properties, along the lines of the following example:

```
{ ...
  [: yellow
     {[: type color]
      [: examples {sun sunflower butter (juice-of orange) gold}]
      [: translations {[: german gelb][: french jaune]}]
      [: Comment
          "Pure gold is colorless but when gold is used in practice it
           is usually in a alloy which gives it a warm yellow color."]
        }]
... }
```

However, this notation would be much less readable. The use of the line of dashes facilitates greatly for the user to orient himself or herself in the text of an entityfile, and it is a minor compromise with the desire for a simple and uniform notation.

The information state in a Leonardo session does not in general contain all the entity descriptions in all the entityfiles of that individual. Usually one only loads those entityfiles into the session that are needed for the tasks at hand.

2.3 The Structure of the Scripting Languages

For the purpose of execution or translation, the elements of GSL scripts can be entities, numbers, or strings. By a particular rule, symbols consisting of a single letter or a single letter followed by one or more digits are considered as variables for the purpose of quantified expressions, and can not be used as entities.

The KRE constructs of sequences, sets and maplets are only used for representing sequences, sets and mappings in the application at hand; they are not used for encoding language constructs. Composite entities are also of course used for representational purposes. Therefore, composite objects in the scripting language are *only* expressed using forms and records. The following major kinds of constructs are used in GSL.

- *Terms* which are KRE forms,
- *Queries* which are KRE forms,
- *Propositions* which are KRE records,
- *Logic formulas* which are KRE forms,
- *Action records* which are KRE records,
- *Scripts* which are also KRE records.

Atomic or composite entities, variables, numbers and strings may occur as terms, and queries subsumes terms. The following are three examples of logic formulas in GSL in order to illustrate these conventions.

```
(implies (and [on x y][on y z]) [on x z])
(forall x physical-object (not [on x x]))
(forall x animal [part-of (head-of x) x])
```

Thus, propositions are formed as KRE records with the predicate as the record composer (i.e., operator) and with terms as arguments. A logic formula is either a proposition or a term that is formed from logic formulas by recursive use of the standard propositional connectives and quantifiers with the notation shown in the examples.

Terms are formed from entities, variables, numbers or strings and are composed recursively using *term formants*. All terms are queries, but queries can be constructed in additional ways by having e.g. a proposition as an argument, as in

```
(those x [larger (population-of x) 5.000.000]
         (members-of european-union))
```

If a composite entity, sequence, set or maplet contains a variable then this variable is supposed to be replaced by its value in some types of computation, for example in the evaluation of the selection condition in this example. Here, `(members-of european-union)` is supposed to evaluate to a set whose members are (entities representing) the member countries of the EU, and the `those` expression obtains a subset of that set using the proposition argument in the obvious way.

2.4 Action Records and Scripts

Action records are used to specify actions that are to be performed when a script or its translation is executed, or an action that has been performed. Thus a command is simply an action record that is used by a user (or by another Leonardo individual) for giving an instruction to a Leonardo individual. A log of what commands have been executed consists of action records that are not (any longer) commands.

The qualifiers of the action may be represented as arguments or as parameters according to the preference of the language designer. (In the record

```
[op a b :tag1 v1 :tag2 v2]
```

we say that `a` and `b` are arguments, and `v1` and `v2` are parameters). As a rule of thumb, arguments should be used for elementary actions and for those objects that are directly affected by the operation, and parameters should be used for objects that characterize how a non-elementary action is performed. The composer of an action record is called an *action verb.*

Scripts are a particular kind of action records where some of the arguments are action records or subordinate scripts, even recursively. Parameters are used for expressing how the script is performed. The following is an example of a script for use by a household robot:

```
[soact [insert key-4 lock-12]
       [turn-clockwise key-4]
       [remove key-4]]
```

In this example, `insert`, `turn-clockwise` and `remove` are assumed to be elementary action verbs, and `soact` is a script composer that simply specifies the sequential execution of the actions (or scripts) given as arguments. Action verbs such as these are of course specific to an application, but the script composer `soact` is independent of application. The Generic Scripting Language specifies a few such script composers, including `soact`, that can be used in all scripting languages in the GSL family.

Parameters may be used for specifying how an action or script is to be performed, but in the case of scripts they may also be used for other information pertaining to the script, as in the following example.

```
[to-achieve [insert key-4 lock-12]
            [turn-clockwise key-4]
            [remove key-4]
            :goal (not [locked lock-12]) ]
```

where the parameter tagged as `goal` is used to specify the purpose of the script. In a planning situation this script may initially be specified using only the goal parameter and without arguments, and then the arguments are added by the planning process.

3 Self-Description Facilities

The self-description facilities are those that are present in a Leonardo individual during sessions (and most of them are present in the persistent manifestation as well), and that serve to characterize the individual and its environment, and to provide access to various aspects of the environment. All self-description facilities are expressed using the Leonardo representation framework and as entities, values of attributes, verbs and scripts.

3.1 Self-description of the Individual and the Startup

Each Leonardo individual has a unique *identifier.* The assignment and maintenance of these identifiers is based on the assumption that there is only one way of creating a new individual, namely by the operation of *reproduction* whereby one individual creates an *offspring.* Each individual has only one parent, not e.g. two. The successive offspring of an individual are numbered from 1 and up, and the identifier for an individual is obtained by appending its offspring number at the end of the identifier of its parent. For example, the second offspring of `lar-001-023` will have the identifier `lar-001-023-002`. The acronym `lar` stands for ”Leonardo Ancestry Root”.

Besides its identifier, each individual also has a *name* which can be assigned more freely. There is no formal requirement that all individuals shall have different names, but it is inconvenient to have several individuals with the same name in a given, local environment [4]. The name of an individual is assigned when it is created, and changing it later on requires a fairly complicated modification of its contents.

The following is an example of an entity-description for the identifier of an individual:

```
---------------------------------------------------------
-- lar-001-023-002

[: type leo-individual]
[: leoname leordo-2]
[: self-location "../../../leordo-2/"]
[: leoprovider "../../../leordo-2/"]
[: leoguru "../../../leordo/"]
[: uses-hostfiles <software madman-roots>]
[: latest-offspring-nr 0]
[: parent lar-001-023]
[: parent-archivenr 6]
```

[4]Except for the use of doppelgänger which will be described in the section on self-modification.

The use of the `type`, `leoname`, `parent` and `latest-offspring-nr` attributes is evident. Each individual keeps track of the number of the latest offspring it has produced. The use of the other attributes will be described further on in this report.

Each individual has several *startup entities* which represent different ways of starting a session. There are different startup entities for running the individual in a Windows context or a Linux context, and for running in different languages and language variants (for example, different implementations of Lisp). Each startup entity also specifies which entityfiles are to be loaded during startup of a session. The following is an example of a startup entity:

```
---------------------------------------------------------
-- minileo

[: type startup-file]
[: contents <minileo>]
[: inprocess main]
[: batname "minileo"]
[: for-os-family windows]
[: lispvariant allegro]
[: kb-included <core-kb>]
[: bootfile "../../Core/cl/bootfuns.leos"]
[: execdef lite-exec]
```

The purpose of the attributes `for-os-family` and `lispvariant` is evident. The `batname` attribute specifies, in the example, that the system shall generate a file called `minileo.bat` that can be invoked in order to start a session with the startup specified here. All the startup files (e.g., `.bat` files) of an individual are collected in the same directory within the individual. The `bootfile` attribute specifies the path to an entityfile that contains not merely data, but also expressions that are to be executed, and that is used to start the session. The `kb-included` attribute specifies what knowledge-blocks (i.e., sets of entityfiles) are to be loaded during startup. The `execdef` attribute specifies what is to be the (initial) command-line executive during the session. Finally, the `contents` and `inprocess` attributes serve simple administrative purposes.

There is also a facility for self-description of each session, which requires an entity that serves as a unique identifier for the session at hand. This entity is a composite entity that is formed using the operator `session:`, for example `(session:  lar-001-023-002 93)`. Each individual maintains a counter for how many sessions have been started in it. The following is an example of a session entity:

```
---------------------------------------------------------
-- (session: lar-001-023-002 93)

[: type leosession]
[: inhost acer3020]
[: configuration minileo]
[: init-leos-files <bootfuns selfdescr self-kb kb-catal software
       madman-roots minileo>]
[: runstart-ms 100740059468]
[: runstart-datetime <2009 3 10 19 11 50 1 t -1>]
[: curtime 14]
```

The purpose of the `inhost` and `configuration` attributes is evident. The `runstart-ms` and `runstart-datetime` attributes specify the starting time in milliseconds and in terms of calendar and clock. (The words ”run” and ”session” are treated as synonyms). The `init-leos-files` attribute specifies the initial systems-related entityfiles that were loaded at the very beginning of the startup process. The `curtime` attribute, finally, contains a counter that is incremented by one each time an action is recorded in the session log, so it provides a simple timestamp mechanism for use within the session.

Within each session there are global quantities whose values are the identifier of the individual at hand, the startup entity that was used for the session at hand, and the session identifier. In the CommonLisp implementation of Leonardo these are represented as the values of the global variables `*my-id*`, `*myconfig*` and `*cursession*`, respectively [5]. In this way it is possible for processes within the individual to find out about in what context they are executing.

3.2 Self-description of Sessions and Episodes

Leonardo individuals routinely build a record of what actions have been executed during a session, from the top level and down to some limited level of detail but not necessarily down to the level of elementary actions. This facility can be used for a number of purposes, including for tracing when new programs and scripts are tested, for “redo” commands during the command loop, and in order to be able to extract and archive a full account of what has happened during an important system demonstration. We therefore consider it as a basic facility that should be an integral part of the core system.

Not all logs of actions during sessions are worthy of being preserved for posterity, however. The logging is therefore done in two steps. The first step is always in place and it creates and maintains a structure consisting of records and subrecords that represent the log. The second step is optional and consists of saving this structure or selected parts of it as persistent entityfiles. Some modifications are also done to the structure when it is archived.

A plain linear log of actions would be impractical, and there are two devices for assigning a structure to it. First, there is a structure of *episodes* and *sub-episodes*. When a session is started, the entity `episode-0000` is set to be its *current episode*. It is also by definition the *top episode* throughout the session. Whenever it is considered appropriate it is possible to create a new episode, e.g. `episode-0001` that is a subepisode of the one at hand, and to make it the current episode. Each episode has an attribute for its sequence of sub-episodes and for its super-episode (except of course `episode-0000` does not have any super-episode). In this way it is possible to give separate identity to the log of some part of a session.

[5]Only the last one of these global variables is actually needed since the entity description for the session contains the information for the two others. They are retained anyway for reasons of convenience and legacy.

Each episode has an attribute `has-chronicle` whose value is a *chronicle*, i.e. a record containing a sequence of action records or subordinate chronicles. The following is an example:

```
[chronicle :clock (session: lar-001-023-002 93)
           :occurs-in episode-0002 ^
     [crefil sessiontest :at-time 152]
     [setk cur-kb :at-time 151 :result cur-kb]
     [dpdf :at-time 150 :result "../../../../Series/caisor/2009/pm-012/"]
     [lam :at-time 149 :result "../../../../Series/caisor/2009/pm-012/"]
     [cocl body :at-time 148 :result nil]
     [loadfil compos-verbs :at-time 147 :result compos-verbs]
     [writef :at-time 146 :result nil]
     [loadf :at-time 145 :result elem-verbs]]
```

A chronicle record has two parameters, `:clock` and `:occurs-in` . The latter specifies the episode within which the chronicle occurs. The `:clock` attribute is used in general for specifying an entity that has a `curtime` attribute and that can be used to provide timestamps for the actions in the episode. The session identifier is the natural choice for this purpose, but one can choose to use another clock for a chronicle if one wishes. The same clock must be used throughout the chronicle, but a sub-chronicle may use another clock.

The arguments of a chronicle record are action records that have been performed within the scope of that chronicle, in reverse order. Every such record is obtained as the record that was used in order to invoke the action, but with the addition of two parameters, namely `:at-time` which specifies the time of the action according to the chronicle's clock, and `:result` containing the action's result record. Result records will be described separately, but essentially whenever an action is performed, the interpreter produces a result record which specifies whether the action succeeded or failed, and sometimes with an indication of a result from the action.

The identifier `episode-0000` is sufficient for identifying the top-level episode and the entire episode hierarchy within a session, but if one needs to work with episodes and chronicles from different sessions then the use of this identifier would lead to name conflicts. Therefore, the separate facility for archiving of episodes and chronicles replaces an episode identifier such as `episode-0002` with the composite entity

```
    (episode: (session: lar-001-023-002 93) 2)
```

in the example above. The reason for not using such entities already from the beginning is merely for ease of reading.

3.3 Catalogs and Self-description for Entityfiles

As already mentioned above, an entityfile can be understood both as a sequence of entities, and as a text file containing entity descriptions for the entities in that sequence. When we use the word entityfile we shall mean the textual representation unless the contrary is stated.

The duality between the persistent and the dynamic representation of entities requires that if an entityfile has been loaded into a session then it shall always be possible to re-generate the entityfile from the information in the form of entities, attributes and values that are present in the session's data structures. This is implemented as follows. For each entityfile there is a specific entity that is the *name* of the entityfile. Its type is, or is subsumed by the type `entityfile`. The following is an example of the entity description for the name of an entityfile.

```
---------------------------------------------------------
-- misc

[: type entityfile]
[: latest-written "2009-03-10/19:11"]
[: contents <misc misc-oper-definers c-ans defleopa defop>]
[: purpose "Miscellaneous small function definitions"]
```

Notice that the name of the entityfile has a *contents* attribute whose value is a sequence of all the entities in the entityfile. The name shall itself always be the first element of that sequence. The operation of re-writing the entityfile from the current information in the session will therefore take the name of the entityfile as its argument, and it makes a loop over the elements of the `contents` attribute and writes the entity-description for each of those entities on the newly generated text file. More about this in the section about the type structure that will follow below, and in the separate report on type structures and ontology in Leonardo.

There is an issue about where entityfiles are to be located in the computer where the individual resides. In principle they shall be somewhere in the directory structure that is the persistent manifestation of the individual, but having them all in one single directory would not be practical, and we prefer to partition them into several directories for easier overview. Moreover, there are some special cases where entityfiles are located elsewhere, for example, a few entityfiles that describe aspects of the host where the system is running. These entityfiles should only be present in one single version that is shared by all the individuals on that host.

Therefore we need to have a facility that specifies, for a given name of an entityfile, where the (textual) entityfile is located. This information can obviously not be kept in the entityfile itself, since it is needed already to find the entityfile.

The solution to this is to use *location entities* which are composite entities whose major purpose is to carry the information about where an entityfile is located. The following is an example of a location entity:

```
---------------------------------------------------------
-- (location: els-designators)

[: type location]
[: filename "../../../leordo-2/Els/els-designators"]
```

This works as follows. Suppose the top-level directory of the Leonardo individual whose name is `leordo-2` is

```
    G:/Aims4/leordo-2/
```

Then there is one specific subdirectory where sessions are started, namely

```
    G:/Aims4/leordo-2/Process/main/
```

This means for example that in a Windows environment, the `.bat` files mentioned above are located in this directory. The entityfile `els-designators` in the example is located at

```
G:/Aims4/leordo-2/Els/els-designators.leo
```

The value of the `filename` attribute is therefore the relative path from the starting directory to the entityfile at hand.

One may wonder why this relative path has to go up above the level of `leordo-2` and then down again. Would it not be more convenient to simply let it have the value `"../../Els/els-designators"`? The reason is because we may have several individuals, similar to `leordo-2`, which are located side by side in one single master directory, which is `G:/Aims4/` in the example. The chosen representation makes it possible for an individual to use the location entities of neighboring individuals without the need for converting the values of their `filename` attributes.

This arrangement makes it possible for a session to load an entityfile if it has already loaded its location entity. Location entities must therefore be loaded to begin with. To this end the individual contains a number of *catalog files*, which are simply entityfiles where most or all of the contents are location entities. The starting point is one specific entityfile that is called `kb-catal` in every individual, and that contains the location entities for a number of further catalog files which in turn contain location entities for ordinary, non-catalog files. Additional layers of catalogs are used in some specific situations.

The startup process therefore always loads `kb-catal`, and then it uses the information in that file to load the other catalog files that are specified in the `kb-included` entity of the startup entity.

3.4 Knowledgeblocks

The number of entityfiles in a Leonardo system is so large that it would be inconvenient to treat all of them as an unstructured collection. They are therefore partitioned into *knowledgeblocks*. Each knowledgeblock is represented as an entity, one of whose attributes is a list of the entityfiles that are members of the knowledgeblock. Moreover, the entity for a knowledgeblock is a kind of entityfile entity, so each knowledgeblock contains one entityfile with information that pertains to the knowledgeblock as whole.

The knowledgeblock structure is important for two purposes: for loading information into a session, and for version management. When a session starts, the Leonardo system first loads a small number of very basic entityfiles and then it loads the *core knowledgeblock*. After that, additional knowledgeblocks are loaded according to the `kb-included` attribute of the startup entity. Loading a knowledgeblock consists of the following steps:

- Loading the entityfile describing the knowledgeblock as a whole
- Loading (recursively) other knowledgeblocks that are prerequisites of the one at hand
- Executing the pre-startup procedure of the knowledgeblock
- Loading a subset of the entityfiles that belong to the knowledgeblock

- Executing the post-startup procedure of the knowledgeblock

The information that is needed for steps 2 to 5 is provided in the entityfile that is loaded in step 1. In particular, the entityfile for a knowledgeblock usually contains the location entities for all the entityfiles in the block except itself. The location entities for knowledgeblock entityfiles are in the designated entityfile `kb-catal`, in standard cases, but they may also be located elsewhere and loaded by application-specific means. In applications where there is a large number of entityfiles with a similar structure one may arrange to generate location entities when needed.

The pre-startup and post-startup procedures are often empty but they are occasionally very useful.

Entityfiles that belong to the knowledgeblock but that are not loaded in step 4 above can anyway be loaded later on, by explicit command by the user or in other ways. Likewise it is possible to load entire knowledgeblocks at later times during the session.

The use of knowledgeblocks for version management is described in a later section.

3.5 Description of the Host Computer

In order to run sessions of a Leonardo individual on a particular computer, it must have two things: the persistent manifestation of the individual, and an implementation of one of the programming languages that this individual can run on. At present this only means Allegro CommonLisp, but the system design is such that other languages can be used on equal terms with the first one. The possibility of having multiple startup entities which was described above, is essential for making this possible.

In addition each session needs to have some information about the host computer on which it is running. For example, if a session wishes to invoke some standard software, such as a text editor or a pdf reader, it must know where this program is located on the the current host. The network name of the host is also sometimes needed.

For this purpose there is one designated, but OS-dependent location where host-specific entityfiles are kept. In the Windows environment this is the directory `C:/Leohost/`. This directory must always contain an entityfile called `software` that contains the access paths for certain standard software. Additional files may be added in this location for use by particular services. The following is an example of an entity description in one such file:

```
------------------------------------------------------------
-- bibtex

[: type os-command]
[: commandphrase "C:\Progra~1\MiKTeX 2.6\miktex\bin\bibtex.exe "]
```

The value of `commandphrase` can be either a string or (to account for more complex cases) a KRE expression. If it is a string [6] and the program

[6]Strings in Leonardo do not use any escape character, so the backslashes in this example denote themselves. If one wishes to use a string containing a double

in question is one that needs a file as an argument, then the value of `commandphrase` shall be written in such a way that one can concatenate its value with the name of, or path of the file and obtain a correct OS command for invoking the program.

The example of an identifier for an individual which was shown earlier, contained an attribute called `uses-hostfiles` where the value is a sequence of symbols. This value shall be a list of those host-specific entityfiles that may be used by the individual in question.

3.6 Description of Available Network and Servers

As has already been described, each Leonardo individual can be activated as a session but normally only one session at a time for each individual, and each session can run an http server whereby the individual can communicate with users (as a complement with command-line dialog) and with other individuals. Since several individuals can run in the same host computer, each individual is associated with a port number for use by its http server. Furthermore, each individual needs to have information about the location and other contact information for other individuals, including both those that are in the same host, and in different hosts in the same local area network, and elsewhere in the Internet.

The system-describing information structure that makes this possible has two main parts: the grouping of several individuals into a *residence*, and the use of *network information*. Some of the network information is represented separately for each individual, but some of it is handled jointly for each residence, so we shall describe residences first.

A Leonardo residence is a directory containing the persistent manifestations of one or more Leonardo individuals as *immediate* subdirectories. It is therefore a way of grouping several individuals together in such a way that they can refer to each other in the simplest possible way, using relative paths for files. The hard disk or other persistent memory of a host computer may contain one or more residences, but often one wishes to keep individuals on a USB stick or other easily mobile memory. In such cases one must define a separate residence on the USB stick.

Besides the member individuals, a residence also contains an additional subdirectory called `Residmap` containing entityfiles with descriptions of the individuals in that residence, as well as some network information that is used jointly by the individuals in that residence.

Entityfiles in a residence map use the following entity types, most of which are self-explanatory:

- `local-area-network`
- `computer-host`, which may be either a desktop or a laptop computer
- `data-carrier`, e.g. a particular USB stick
- `leo-residence`
- `agent-descr` and `indiv-loc-descr`: entities that are introduced for providing additional information about a Leonardo individual.

quote inside it, one constructs it using a KRE expression.

The following is an example of an entity description for a local area network (LAN) entity:

```
---------------------------------------------------------
-- lan-home-erisa

[: type local-area-network]
[: has-ip-nr 81.231.167.19]
[: assigns-ip-nrs {[: guardian 192.168.0.5]
        [: mediacomputer 192.168.0.6]}]
[: assigns-ip-wifi {[: acer-3020 192.168.0.3]}]
[: forwards-ports {[: 80 192.168.0.5] [: 99 192.168.0.5]
        [: 20 192.168.0.6]}]
[: has-messageport 99]
```

The user chooses a name for the LAN at his own discretion. The attributes for the LAN entity reproduce some of the configuration information in a network router, including the external IP number of the LAN and the assignments of local IP numbers to computers in the LAN. (The attribute `has-messageport` will be explained below). These computers are identified by their network names, which should be chosen as the network names in the sense of the operating systems, and in lowercase mode only. These computers must have unique names in the information context at hand. The following is an example of an entity description for a host computer:

```
---------------------------------------------------------
-- acer3020

[: type computer-host]
[: has-residences {aims4}]
```

Miscellaneous other information may also be attached using other attributes, but the only obligatory information about a host computer is what are the names of residences in it. Residences on different hosts may have the same name since they are always referred to using composite entities like in the following example:

```
---------------------------------------------------------
-- (resid-of: acer3020 aims4)

[: type leo-residence]
[: homedir "C:/Aims4/"]
[: hosts-individuals {lar-001 lar-001-023}]
```

The entity `(resid-of: acer3020 aims4)` represents the residence that is called `aims4` and that is located on the computer called `acer3020`. The `hosts-individuals` attribute specifies the identifiers of the individuals in this residence. Finally, since residences can also be located on data carriers, such as USB sticks, we need entities for data carriers like in this example:

```
---------------------------------------------------------
-- usb-cruzer-4

[: type data-carrier]
[: has-residences {aims6}]
```

Data carrier entities can be used as the first argument of the operator `resid-of:`, just like host computer entities.

Finally, communication between individuals requires that there is shared information about the location and modes of access to each individual. In principle this should be represented as additional attributes for the entities representing each individual, but these entities (such as `lar-001-023` in the example) have already been assigned a type and corresponding attributes in the self-description for the individual that was described above. Therefore we introduce a composite entity for each individual that provides a server, as in the following example.

```
----------------------------------------------------------
-- (indiv-loc: lar-001-023-002)

[: type agent-descr]
[: leoname leordo-2]
[: in-residence (resid-of: usb-cruzer-4 aims6)]
[: uses-port 8091]
[: uses-http-address "http://localhost"]
```

Notice that the information that is already stated with the entity `lar-001-023-002` is a part of the self-description of that individual, and that it is normally only available within a session of that individual itself. The entity shown here is part of the residence map, so it is used by all the individuals in the residence, and in fact it may also be included in other residence maps so as to make it possible for individuals in different residences to communicate with each other.

For some purposes it is useful if a LAN acts as an agent in itself, and in particular one may wish to send a message to a LAN, for example to request an update in its structure, or in order to send a message to a particular individual in that network whose communication port is only available inside the LAN. The `has-messageport` attribute of the LAN entity specifies a port that should be used for these purposes. If a LAN specifies a value for this attribute then it should forward incoming traffic on that port to one of its individuals, and that individual shall be prepared to play the role of high-level LAN gatekeeper as it receives and honors inter-LAN messages.

3.7 Configuration Information

The entity descriptions that have been shown above do not specify which data carrier is located in which computer host, and which computer host is located in which local area network. This is by intent since the location of data carriers and of computer hosts may change from one time to the next. Therefore there is a entity type `leo-configuration` where each entity in this type contains one specification of the locations of data carriers and host computers. More about this in the section about self-modification below.

3.8 Additional Topics

It is intended to add a section on the type system for entities in a later version of this report.

4 Autonomous Action

The characteristic property of an intelligent autonomous agent is that it has a capability for autonomous action. The self-description facilities that have been described above are intended to be a comprehensive basis for autonomous action in Leonardo individuals, besides their other uses. The following are the most important aspects of the autonomous action capability.

- Scripts and other specifications for how to perform actions
- Restrictions on the applicability of actions and on the proper way of carrying them out
- Agenda for forthcoming actions
- Agenda for actions that are scheduled to be performed repeatedly
- Goal-directed invocation of actions
- The history of actions that have been performed by the individual at hand, as well as information about the context of each action.

These are topics for additional, forthcoming reports. Here we shall only discuss one example of an architecture for autonomous action with the aim of showing how it may may be related to the Leonardo architecture.

4.1 The Soar Architecture for Human Cognition

The Soar architecture was developed during the 1980's by Alan Newell, John Laird and Paul Rosenbloom as a candidate unified theory of cognition. It has been extended, implemented as a widely distributed software system [7], used for a number of applications, and it has an active user community. It has also inspired research on intelligent autonomous agents, case-based systems, and other branches of artificial intelligence. In this subsection we shall briefly review the main ideas in Soar and show how they can be implemented in the Leonardo framework.

Soar makes a distinction between two levels of memory: long-term memory (LTM) and working memory (WM). Long-term memory is persistent and accumulative: information is only added, never removed. Information is pulled from LTM or from other sources to WM when the system encounters a specific task.

Working memory is organized as a hierarchy of *states* each of which contains a set of *objects*; each object has a set of *attributes* with corresponding *values*. Equivalently, each state is a set of triples consisting of *identifier*, attribute, and value. Such a triple is called an *augmentation*. The following is an example of a simple state in Soar's working memory and in the representation that is used in the implementation of Soar:

```
(s1 ^block b1 ^block b2 ^table t1)
(b1 ^color blue ^name A ^ontop b2 ^type block)
(b2 ^color yellow ^name B ^ontop t1 ^type block)
(t1 ^color gray ^name Table ^type table)
```

[7]`http://sitemaker.umich.edu/soar/home`

This describes a state `s1` containing three objects: a gray table and two blocks, one block being blue and the other one being yellow. The symbols preceded by a circumflex character represent attributes. The symbol `t1` is the unique identifier for the table object that is used internally in the system, and the `name` attribute specifies a name for it that may be used externally.

Identifiers and attributes must be atomic symbols and can not be composite expressions, and values must be identifiers, strings or numbers. Composite structures such as sets, sequences and records can however be represented by introducing additional, auxiliary objects.

There are three types of long-term memory: procedural memory containing *rules,* semantic memory containing objects of the same kind as are used in the states in working memory, and episodic. Episodic memory is a record of past situations, actions and events; it may be loosely compared with the use of archived sessions and episodes in Leonardo.

A rule is written on the following general form:

```
sp {rule*name
      (condition)
      (condition)
      ...
      -->
      (action)
      (action)
        ... }
```

Besides its states, working memory can also import rules and past episodes from long-term memory and it can receive information from *perception.* This information is used by a *decision procedure* which results in the selection and the execution of *actions.* In fact, it is the decision procedure that retrieves information from long-term memory and introduces it into working memory.

The decision procedure is the primary “engine” in the Soar architecture. It works in five steps: *input, elaboration, decision, application,* and *output.* Input consists of representing recently arrived perceptual information as working-memory structures, in particular in the current state that is maintained in WM. Elaboration consists of finding rules whose conditions match the current state and of executing their action parts, thereby extending the contents of the current state with additional augmentations. (This may be additional augmentations for existing identifiers, or it may introduce additional ones). Notice that the “actions” at this point merely modify working memory, either by drawing conclusions or by importing additional information from LTM. Removal of augmentations is possible but must be used with caution, since then the result of the elaboration phase is dependent on the order in which the rules were applied. The elaboration phase ends when there is no more applicable rule.

One of the possible results of an action in the consequent part of a rule is a *suggestion* for an *operator* to apply, that is, something that the system will do in the application step of the decision procedure. When the elaboration phase finishes, one possibility is that there is exactly one suggestion for an operator, and that the various applicability conditions for this operator are satisfied. In this case it may be applied at once.

However, there may be a number of complications, for example:

- Maybe there is no suggestion for an operator
- Maybe there are several suggestions for operators
- Maybe there is just one such suggestion, but the operator in question just can not be applied in the current situation
- Maybe it is not possible to determine whether the operator can be applied, since the required information is missing
- Maybe there is just one suggested operator, but the information that has been collected in the current state includes predictions about the direct and indirect effects of applying the operator, and some of those effects are very undesirable

Situations such as these are called *impasses* [8] and they are important in the Soar architecture. Whenever there is an impasse the decision procedure will refrain from applying an operator, and instead it will create a new, subordinate state and invoke the same sequence of five steps on the substate. The substate is initiated with a description of the situation at hand. For example, if there were several suggestions for an operator then the substate will contain these and it will be given the task of picking one of them. Rules that are relevant for the goal of choosing between candidate operators can be used in the elaboration phase of the decision procedure for the substate.

It is natural, but not necessary to make use of goal-directed behavior in the operation of a Soar system. If the system is given a task that is expressed as a concrete goal, such as "find the e-mail address of Sven Svensson" [9] a plausible Soar implementation will set up a state containing the goal "the e-mail address of Sven Svensson is known" together with other relevant pieces of information, for example, information whether a computer with an Internet connection is available in the present situation. The elaboration phase of the resulting decision procedure will suggest operators that may achieve this goal, and the application phase will perform one of them.

More generally, it is easy to see how an impasse may be processed if the impasse state contains a description of the goal that needs to be achieved; it is less obvious what to do in an impasse otherwise.

Impasses are important for an additional reason, besides being the condition that causes a recursive call of the decision procedure: they are also the point where learning takes place in the Soar architecture. The processing of an impasse begins with an initial state and ends with an outcome; this means that if a similar initial state is encountered at a later time then a similar outcome will be relevant. The operation of *chunking* analyzes the process following an impasse after it has finished, identifies what information was essential for the outcome of that process, and packages the result as a rule-like structure that is stored in long-term memory and that can be retrieved and invoked by the system at later times.

[8]Swedish: dödläge

[9]This is one example of the fuzzy borderline between actions and goals. Literally speaking, the request seems like a command to perform the action "find" with a particular argument. However, unless the system has already a procedure for finding the email address of a person, the appropriate interpretation will be "arrange to get to a state where the e-mail address of Sven Svensson is known", thereby stating a goal rather than an action.

Chunking was the only form of learning in the original Soar architecture. Additional forms of learning have been added more recently, such as reinforcement learning.

4.2 Other Cognitive-System Architectures

It is evident that the Leonardo architecture can readily represent the basic form of the Soar architecture that has been described here. However, basic Soar does not solve all problems, and there are a number of other things that one would also like to include in a facility for autonomous action. The introductory paragraphs of the present section mentioned some of these, including the use of an agenda for pending actions, the use of standing goals, and the use of action planning. Although Soar systems contain some of these, it is important to also look at other approaches where similar problems have been addressed, for example from research on case-based planning.

Notice also that the basic Soar architecture does not define a structure for interleaved deliberation and action. Consider a real-world robotic system that receives a task assignment, cogitates in order to make a plan for how to perform the task, executes the first action in the plan (an "operator" in Soar terminology), cogitates again because the outcome of that action was not the expected one, and so forth. There is then a difference between operators that extend the current state in the Soar decision procedure, and operators that are performed in the real world. On a sufficiently abstract level one may not need to make any distinction between them, but in an actual system they must of course be treated very differently.

Finally, the basic Soar architecture assumes that information is accumulated into, but never removed from long-term memory. For a system that operates for a certain period of time it will be important to be able to remove information from memory when it has become irrelevant.

5 Systemwide Configuration Facilities

One important capability of a software system is that it should be able to reconfigure itself so as to adapt to requirements by the user or by its environment. Modern software, including operating systems contain many examples of this capability; sometimes so many that users are overwhelmed by the resulting complexity.

The basic design of Leonardo is very well suited for implementing configuration facilities, since the overall structure of the system is defined in terms of a data structure consisting of entities and their attributes. However, when configuration facilities are introduced into the Leonardo system, our interest does not lie in maximizing the number of such facilities and the number of features in them. Instead, we are interested in identifying design solutions that have the required flexibility but which also have a clean and simple structure that avoids complexity, and which in particular are correctly located in the overall system architecture. In this report we shall only describe one configuration facility since it is a good illustration of these general considerations.

5.1 Multi-Language Configurability

Many software systems provide the capability of using several languages [10] when presenting information to the user. This capability is a mixed blessing in practice since it is not implemented throughout the entire software on a given computer. Consider, for example, the situation where the operating system of a PC operates in Swedish language and the user wishes to install the software for an attached device, such as a printer, for which the installation instructions are only provided in English. If these installation instructions specify what operations need to be done on the operating-system level, for example using the standard menues, then they will of course refer to the English-language variant of the operating system. This leaves the user with the task of figuring out which menu options in Swedish language correspond to which options in English.

Other problematic situations arise if a guest who does not know Swedish is invited to use that computer, or if the user of the computer wishes to implement multi-language software herself. In principle she needs to have a separate computer for each of the target languages in order to be able to test that software in a complete manner.

These examples suggest two important design decisions. First, unlike what is the case in contemporary operating systems, it should always be possible to change communication language by a simple command. This principle applies both to the operating system and to all other software that runs under it. Secondly, there should be a uniform design for the definition and use of multiple languages at all points in the code where messages to the user are produced.

In the case of Leonardo, one of our long-term goals is to extend the design in such a way that it can be used for the design of the future top-level software in a computer (regardless of whether this software will be called an operating system, or something else). We have therefore defined a facility for multi-language configurability that is set up for system-wide use and which is gradually being introduced in all parts of the current system.

The basic idea is as follows. Consider a piece of code that contains the output of a message on the operator screen in a command-line dialog situation, for example (attribute assignments omitted, actual code in CommonLisp):

```
-------------------------------------------------------
-- example-func

(defun example-func (n)
    (princ "The square of the argument is: ")
    (princ (* n n))
    (terpri) )
```

In order to *prepare* this function for multi-language use it is rewritten as follows:

```
-------------------------------------------------------
-- example-func
```

[10] In this section we use the term “language” in the sense of natural language, for example English or German.

```
(defun example-func (n)
    (princ (phrase :/-1))
    (princ (* n n))
    (terpri) )

@Phrases/English
:/-1 The square of the argument is:
```

The immediate effect of this is that the `Phrases/English` property of the entity `example-func` obtains as value a string consisting of one line, containing both the phrase number and the phrase in question. Two additional things happen when an entityfile containing this definition is loaded into a session. First, the construct `:/-1` in the function definition is interpreted as a macro and is replaced by the symbol `*/-1-example-func` as an argument to the function `phrase`. Secondly, at the end of the entityfile loading operation, the loader inspects all the entities in the newly-loaded file and checks if they have a value for the `Phrases/English` property. If so, it processes the property value line by line and assigns, in the example, the defining phrase as the value of the `in-english` property for the entity `*/-1-example-func`, using the identifier for the entity where the property is located in order to obtain an unambigous symbol. Definitions for other languages besides English are treated in the same way. There can of course be several phrases numbered 1, 2, 3, etc. for any given entity and language. With this design there is no conflict when different entities use the same phrase numbers.

The function `phrase` is defined so that it looks up the phrase definition of its argument for the language that is presently set as the working language in the communication at hand. With this design it is straightforward to represent the multi-language aspect when writing program code and it is easy to see what text will actually be output at a given point in the program.

Usually it is convenient to first write a program using just one communication language, which does not necessarily have to be English, and to add other languages afterwards. It may also be useful to use an automatic translation program for these phrases and just make a manual correction when needed. Therefore the system has been set up so that it is not necessary to have all the language variants together with the entity using the phrase; they can also be assigned elsewhere.

In this example the output for the phrase consists of a fixed string. However, the function `phrase` is defined so that it first obtains the phrase definition for the current communication language, and then applies the script interpreter for the DSL (document scripting language) on this definition. Parameters for this DSL script can be provided as additional arguments to `phrase`. This makes it possible to have boilerplate phrases where elements are inserted within the phrase, and to define phrases with conditional expressions and repetition.

It is intended to introduce these conventions also in the definitions for DSL, and in particular for the generation of web pages. We do not foresee any difficulties in doing this, but it has not yet been done.

6 Controlled Self-Modification

Many modern software systems contain facilities for receiving and installing updates automatically. Many software development projects use version management systems that administrate successive versions of software modules that have been developed and amended by different members of the development team. These are two examples of how a software system is able to *modify itself* albeit according to instructions that are ultimately due to human intervention.

In addition, several branches of A.I. research are developing techniques for automatic adaptation of previously used methods or scripts whereby they become applicable to new situations that are somewhat different from where the same script has been used before. The chunk learning facility in the Soar architecture is one example of this.

6.1 Self-Modification and Autonomy

These considerations suggest that in a self-describing system architecture like Leonardo, it should be important to have a systematic framework for self-modification as a part of the basic design. We only use the term "self-modification" for referring to long-term changes of the structures and programs in a software system, and in our case in a Leonardo individual. Changes that are only for immediate use, such as the dynamic modification of an action script in response to some problems during its execution, will be considered as an aspect of autonomous action which was briefly mentioned in section 4.

Self-modification in this sense can be *directed* (in the sense of "under the direction of someone") or *autonomous.* Installation of software modules and of updates are examples of directed self-modification, along with the use of version management systems and other software maintenance tools.

Autonomous self-modification may be of interest as one aspect of autonomous behavior in general in a software system. For example, if an autonomous real-world robot is organized so that it first pseudo-executes action plans or scripts in a simulator before it attempts to execute them in the real world, and if it happens repeatedly that the outcome that is predicted by the simulator does not agree with the real-world outcome, then this may be a reason for changing specific pieces of code in the simulator. Similarly, a script for searching a variety of sources on the Internet in order to retrieve a particular kind of information may be modified based on the repeated experience with the script at hand.

One obvious aspect of autonomous self-modification is that it is explorative: the proposed new program stub, script, "method", "rule" or "chunk" may or may not turn out to be an improvement over the existing ones. Up to a point it may be appropriate for a system to accumulate such new items and merely label them with a measure of their adequacy or inadequacy, but in the long run it will also be necessary to remove irrelevant information from memory. A version management system may then be very useful as an intermediate solution between full retention and full removal: information that has been archived as an earlier version is removed from the field of attention so it does not disrupt current operation, but it can be recovered if there is a need for this. – Additional uses of version management in the context of autonomous self-modification will be discussed in section 7.

Another aspect of self-modification is that there is a danger of changes that disrupt the software individual doing it. There are several ways of meeting this danger, for example using a "sandbox" technique for executing new programs and scripts, or by arranging individuals in a residence so that one of them can help another one to recover if it comes into distress.

A software design for self-modification should therefore contain a design for version management and for other aspects of controlled self-modification in its lowest layer. This is motivated in pure software systems terms, since version management is used for all kinds of software development, but it is also relevant for the implementation of self-modification. This may be disappointing since version management is such a mundane issue compared with the lofty goal of autonomous self-modification, but it is natural if one wishes to build a system from bottom and up.

6.2 The Version Archive in Individuals

Our framework for version management is based on the same kind of information structure as has been used in the previous sections, and it makes active use of some of the entity-types that have been described there. It is organized in the following major parts:

- A *version archive* within each individual
- A *version exchange mechanism* within each residence
- A facility for *remote exchange of version archive information*
- A facility for *distribution and reception of amendments* to the current information state of an individual.

One purpose of the amendments facility is disseminate information about new versions of knowledgeblocks that have been made available for individuals to download. It can also be used for distributing minor patches to archived entityfiles and information of a temporary character, such as the current location of mobile memories in computer hosts, and the location of computer hosts in local area networks.

The purpose of the version archive in an individual is to preserve information about how the contents of entityfiles have changed. This is done on the level of the individual entity-descriptions, unlike the "file compare" mechanism of conventional version management systems.

It works as follows. Each archiving operation operates on one specific knowledgeblock and preserves a copy of those entityfiles in the knowledgeblock where there has been some change of contents since the previous archiving of the same knowledgeblock. The archiving operations of an individual are considered as a linear sequence, i.e. there is no possibility of concurrent lines of archiving within one individual. An *archive point* is a Leonardo entity representing such an archiving operation.

Technically, archive points are represented as symbols `ap-0001`, `ap-0002`, etc. and as subdirectories with the same names in a designated directory called `Savestate` in the persistent manifestation of the individual. Each subdirectory contains those entityfiles that were archived at its archive point. In addition, each individual has an entityfile called `syshist` containing entity descriptions for all the archive points, including which knowledgeblock and which entityfiles have been archived in each of them, as well as a timestamp.

The version archive is useful for recovering older versions of a given entityfile if need be, but it is also the basis for the exchange of updates between individuals. To this end it maintains the information, for each entity in each archived entityfile, about what was the latest archive point where this entity obtained its current value. For example, if the entityfile `colors` was archived at `ap-0010`, `ap-0020` and `ap-0030` and the entity `yellow` in that file had one value in the archived version of `colors` in `ap-0010` and a second value in the later two archived versions, then the `latest-rearchived` attribute of `yellow` is `ap-0010` in the first archived version and `ap-0020` in *both* the later two archived versions of the entityfile.

6.3 The Version Exchange Mechanism in Residences

Although the entityfile archive within one individual is strictly linear, there is no such linearity when the archives of several individuals are considered jointly. If a particular entityfile is maintained in several individuals and these make updates to the entityfile independently of each other, then it may be a nontrivial operation to reconcile the changes and construct a new version of the entityfile that combines the changes of the participating individuals.

Such reconciliation (often incorrectly called “synchronization”) of entityfiles can however be simplified if the concurrent changes refer to different entities, and by using the value of the `latest-rearchived` attribute.

The implementation of this operation makes at present the assumption that each individual designates a specific other individual in the same residence as its *guru*, and that changes in entityfiles flow from a guru to its clients and from a client to its guru, but not in other directions. In a simple case the residence contains one single guru which has all the other individuals in the residence as its clients. The guru has the common software repository in the residence, but each client may develop its entityfiles independently e.g. in a period of software development.

The entityfile `syshist` in a client contains one additional attribute for each archive point that is listed there, namely, an attribute called `agreement-ap` whose value (if present) is an archive point in the corresponding guru for the same knowledgeblock. This attribute expresses a cross-connection between these two archive points in the respective individuals, so that the contents of all the entityfiles in that knowledgeblock had equal contents [[11]] at the respective archive points.

Notice that this equality requirement applies to all entityfiles that were defined in the knowledgeblock at those times, and not merely those that were archived at those times. For example, if the knowledgeblock at hand contains the two entityfiles `colors` and `sounds` and it was archived at archive

[11] apart from administrative information that is specific to the individual, such as the value of the `latest-rearchived` attribute.

points `ap-0010`, `ap-0020` and `ap-0030`, like above, the entityfile `colors` was archived at all three archive points, but the entityfile `sounds` was only archived at the first two points because all its entities were unchanged at `ap-0030`, then the archived version of `sounds` at `ap-0020` is considered to be logically present at `ap-0030` as well for the purpose of comparison with the corresponding archive point in the guru [12].

The combination of the `latest-rearchived` attribute for all entities and the `agreement-ap` attribute for archive points makes it possible to merge changes in a clear and well-defined way. This operation is performed in a client and operates on a specific knowledgeblock that is defined both in the client and in its guru, and that has been archived in both, so that the current versions of the entityfiles are equal to the most recently archived versions. The purpose of the operation is to identify any differences between the current versions of the given knowledgeblock and to update them if necessary so that they are then equal. If an update is required on one or both sides then a new archive operation is performed there after the update. Finally, if there were any changes, the `agreement-ap` attribute of the most recent archive point for this knowledgeblock in the client is reset so as to refer to the most recent archive point for the knowledgeblock in the guru.

The realization of this operation in the client is straightforward. For a given knowledgeblock, identify the most recent archive point for this knowledgeblock that has a value for its `agreement-ap` attribute. This archive point will be called the *client base*; the value of its `agreement-ap` attribute will be called the *guru base*. Now consider all the entityfiles that belong to this knowledgeblock according to the `syshist` file of the client or the guru, and all the entities in each entityfile. For each entity, compare the values of its `latest-rearchived` attribute with the base on the respective side. If both of them are equal to, or precede the base, then there has not been any change. If both of them succeed the base on its side, then there has been concurrent change in this entity for the purpose of the merge operation. If one of them is equal to or precedes its base and the other one does not, then there has been a change on one side only.

For each entityfile, if none of the entities in it has a change then there is no change in that entityfile. If all the changes in it are on the same side (client or guru) then the file will need to be copied from client to guru, or from guru to client. If there are some client-side changes and some guru-side changes, but no concurrent change, then it is straightforward to construct a new version of the entityfile that contains the most recent version of each entity, and to provide it to both the client and the guru.

If there is also one or more concurrent changes in an entityfile, then the operation must produce a version of the entityfile that contains both the competing current versions for concurrently changed entities, as well as of course the appropriate entity description of any entity with single-side change or without change. This entityfile is then referred to the user who has to reconcile the differences manually.

Notice that up to the point where a concurrent update has been identified

[12] One may then wonder what happens if an entityfile is removed from a knowledgeblock. The answer is that the entity description for an archive point in the entityfile `syshist` contains both a list of the entityfiles that have been archived in the archive directory for that archive point, and the larger list of all entityfiles that are logically present in the knowledgeblock at that archive point.

for one particular entity, all this processing is done on the entity-description level, using archive points that occur as the values of attributes. A file comparison, in the sense of a matching of two texts, may sometimes be useful if there has been concurrent changes in a large property containing e.g. a script or a piece of program of some size, but it is a rare occurrence.

Since this operation is done in the client, it can immediately perform required archiving operations on the client side. If some entityfiles must also be changed on the guru side then it is possible for the client to change the contents of files in the guru [13], since they are located in the same residence, but it would be inappropriate to let the client also do the guru-side archiving operation. There are two possibilities for realizing it: either the user starts a session with the guru and commands the archive operation, or guru and client are running continously as server agents and the client sends a message to the guru requesting an archive operation.

The present implementation requires that each individual has one single guru. It is straightforward to extend the design so that the choice of guru is specific to each knowledgeblock in the client, but this has not been done so far and is not considered as a priority.

6.4 Remote Distribution of Version Archive

If several individuals are permanently located in the same residence and are going to use a particular knowledgeblock without any individual differences, then there is no reason for them to have their own local copy of that knowledgeblock. It is sufficient to maintain one copy of it, for example in their shared guru, and to let sessions for each individual load the knowledgeblock from the common copy.

Residence-level version exchange is however useful for software development, where a new facility can be developed in one (or more) of the individuals without disturbing the others, and in order to be uploaded to the common guru when a satisfactory new version has been obtained.

In addition, there is a need for remote exchange of versions and of version archive information. Consider in particular the case where a distributed application requires that Leonardo individuals using the same software operate on several host computers, and one wishes to distribute software updates to them in a systematic fashion.

The concept of a *doppelgänger* is introduced for such situations. A doppelgänger of a Leonardo individual is another individual that is located in another residence and has the same parent individual and the same name (in the sense of the `leoname` attribute in the self-description) but of course another identifier. The doppelgänger shall also be a “mirror” of the original individual in the sense that all the knowledgeblocks in it are copies of the same blocks in the original individual.

For example, the following might be the entity-description of a doppelgänger of `lar-001-023-002` which was used in section 3.1 above:

[13] It would be a cleaner design to let the client place the updated files in a mailbox where the guru can pick them up, but in practical operation this does not make a difference.

```
------------------------------------------------------------
-- lar-001-023-011

[: type leo-individual]
[: leoname leordo-2]
[: self-location "../../../leordo-2/"]
[: leoprovider "../../../leordo-2/"]
[: leoguru "../../../leordo/"]
[: uses-hostfiles <software madman-roots>]
[: latest-offspring-nr 0]
[: parent lar-001-023]
[: parent-archivenr 6]
```

However, doppelgänger are used in particular for guru individuals which do not usually have a value for the `leoguru` attribute.

Suppose in the example there is going to be three hosts, each containing a residence with one or a few active individuals. One of the hosts is designated as the primary one and will be the source of software updates. The guru in the primary host is considered as the primary guru, and the gurus in the other two residences are doppelgänger [14] of the primary one. The contents of a doppelgänger are updated automatically whenever there is a change in the corresponding contents of its original. Client individuals in each of the residences may load entityfiles from the guru in their respective residence, or download their own copies using the intra-residence version exchange mechanism.

One of the features of this design is that it is possible to move the persistent manifestation of a client individual from one residence to another without any changes whatsoever in the directory structure and the files representing the individual. Relative access paths to entityfiles in the guru will work correctly as access paths to the doppelgänger, since the original individual and all its doppelgänger have the same name. The same is true for the `agreement-ap` attribute used in version exchange.

In particular, when additional individuals are needed in the application, it is possible to breed them in the primary residence and then move them to the residence where they are to operate. It is also possible to breed them in the subsidiary residence, but this results in an exceptional situation where a change is first made in a doppelgänger guru and has to be mapped back to the primary guru. Such situations have to be handled with due care.

The automatic update of doppelgänger has not yet been implemented at the time of writing this report but it ought to be straightforward since it is a one-way update operation, except for the occasional case of breeding new individuals in a subsidiary residence.

The structure that has been described here, using clients, gurus and doppelgänger is a relatively rigid one. Specific types of applications will require different extensions. The most general case is of course if any individual may exchange scripts and other information with any other individual, but some kind of structure on the update activity will always be necessary in order to protect the integrity of the software in each individual. This may be a communal structure where the individuals participate, like the one just described, or it may be a structure that is internal to each individual and

[14]The word doppelgänger is the same in the singular and in the plural.

whereby it protects its own integrity, but at least one of those is needed. The next section will briefly discuss one direction that the extensions can take, but this should only be seen as an example.

6.5 Distributed Software Development

One particularly important case occurs if several persons are developing software for a joint system of Leonardo individuals, and where they use separate host computers. (If they work on the same computer or on computers sharing a file system then the guru-client structure in a single residence may be appropriate). The question is what version management discipline will be appropriate for such distributed software development.

There are two obvious possibilities: either to emulate the single-residence structure across network connections, or to arrange that each developer has his or her own residence with its own guru and client structure, and to arrange for the respective gurus to exchange information. The first method is simple and easy to implement. The second method is more complicated, but may be preferable if a developer often works off-line and wishes to maintain his own guru-client structure, for example in order to try different development directions, or even for experimenting with independent learning and knowledge acquisition in each one of several client individuals. However, neither of these alternatives has been implemented yet.

6.6 The Distribution and Reception of Amendments

The version management mechanism that has now been described is useful for archiving software changes in a systematic way. However, it has two limitations: it is a bit impractical when minor changes need to be circulated, and it does not define the mechanism for bringing required changes to the attention of software individuals.

The amendment facility responds to both of these needs. It works as follows. Each guru maintains a set of *amendments*, each amendment being an entity that describes an action to be taken or a change to be made in an individual session if a particular knowledgeblock has been loaded there. Amendments also specify conditions for their applicability, in particular with respect to which version they require for the knowledgeblock(s) that they update, or depend on.

Technically, each amendment is a script in the Generic Scripting Language, including information about when it is appropriate to execute that script, and about the expected results of doing so.

When a knowledgeblock is loaded into a session then the amendments that pertain to that knowledgeblock shall also be loaded, and executed if applicable. This applies both if the knowledgeblock is loaded when the session is started, and if it is loaded later on in the session.

The most straightforward type of amendments are those that request a particular, additional entityfile to be loaded, typically an entityfile that is only present in the guru and not in the client. This makes it possible to add material to a knowledgeblock in a quick way and without having to update its local copies in individuals. The new entityfile may be incorporated as a normal entityfile in its knowledgeblock the next time a major archiving operation is performed.

Other simple uses of amendments are for minor corrections to the self-administration information in individuals, since this information is maintained locally and should not be modified by entityfiles in ordinary knowledgeblocks that can be loaded from the guru.

A further use of amendments is for distribution of the current configuration information. The attachment of data carriers to computer hosts and of computer hosts to local area networks can change from one session to the next, and it is information that shall be shared between the participating individual sessions. Therefore, the definitions of alternative configurations is maintained by the guru in each participating residence (including doppelgänger gurus), and the amendment file is edited so that it specifies which of the configurations is the one that operates at present.

In fact, the configuration information for client individuals within one residence may instead be collected bottom-up, if one so wishes, by allowing each individual session to identify where it is located and to communicate this to the local guru which in turn informs the clients that have registered previously. However, there is still a need for some preloaded configuration information, at least for specifying the IP addresses of participating local area networks.

Finally, one important use of amendments is to specify for a client individual that one of the knowledgeblocks that it maintains in a local copy has been changed in the guru. In such cases the client shall first check whether it has made any changes to the contents of that knowledgeblock, and then take appropriate action for downloading or for reconciliation of changes. If the local contents have changed then more care must be exerted before executing the amendments, for example, by checking with the user.

In fact, the kernel of the Leonardo system maintains an attribute called `changed-since-archived` for entityfiles. This attribute shall have the value `nil` when the entityfile has just been archived, and the value `t` if some of its contents have been changed or, more precisely, if the persistent manifestation of this entityfile has been rewritten since it was last archived.

Fully general use of amendments for performing changes in client individuals may be problematic since it may duplicate, or even interfere with the version management that was described in the previous subsections.

7 Autonomous Self-Modification

Autonomous self-modification must be based on two pillars: autonomous action in general, and controlled self-modification. It is therefore natural that a detailed discussion of this topic must belong to a report on autonomous action that succeeds and builds on the present report. However, some considerations may be made already at this point, and in addition to the remarks at the beginning of the previous section.

A pivotal question must be what can be gained by autonomous self-modification. There is an abstract interest in it as an important component of intelligence as we know it in humans, but a concrete resulting advantage is important both in order to motivate and to focus the design.

The essence of the matter is, in our view, that it concerns *self-modification of the behavior* of the individual or agent that is concerned. This can only be achieved by the individual modifying its own information state, but this is a technical aspect and is therefore a secondary consideration.

Self-modification of behavior appears to presuppose an important design principle, namely, that *actions that are performed by the system at hand are characterized both by a goal and a script.* The goal is a characterization of the desirable state of affairs after the action has ended; the script is the sequence of subactions that will be performed, or attempted, in order to carry out the action. Sometimes a user may state a goal and leave it to the system to select the script, and sometimes the user may specify the actions on some level of detail and leave it to the system to infer what the user "really" wants. Finally, some goals may be standing goals that serve as initiators or as constraints for a number of specific actions.

The importance of combining scripts and goals was illustrated in the section about the Soar architecture, where we suggested that an impasse can best be processed if it contains, or is associated with a specification of what is to be achieved. The combined use of scripts and goals is of course well-known in several parts of artificial intelligence, including algorithms and systems for action planning, intelligent agent architectures, and case-based reasoning and planning. At present we focus on its use in the context of autonomous self-modification.

The learning mechanism in Soar leads to the formation of new "chunks" of behavioral knowledge that can be used at later times when the system encounters a similar problem. It can be seen as self-modification in the simple sense that additional information is stored in Soar's long-term memory, but the more important aspect is that this improves the system's ability to choose and to use an appropriate script, or action plan, in later situations.

If the assumption of a combined goal/script view of actions immediately suggests at least some kinds of autonomous self-modification, the converse is also true: if a system is only defined in terms of scripts and of program modules on different levels of granularity, then in the absence of a notion of goals it is difficult to see how autonomous self-modification may come into play. There is of course the possibility of automatic tuning, performance optimization of certain scripts, partial evaluation of interpreters on different levels, and compilation of program modules to lower level operation sets. All of these may be very worthwhile, but it is not natural to consider them under the heading of self-modification of behavior.

Since the goal/script view of actions is arguably a prerequisite for autonomous self-modification in a software system, the next question will be whether it also enables other kinds of behavioral self-modification besides the introduction and adaptation of scripts that lead to specific goals. One interesting possibility is for *goal revision,* which may work as follows. Consider an autonomous agent that operates in an environment where it performs tasks within the framework of its own permanent goals, or policies, but where the environment is a dynamic one in the sense that there may be chains of cause and effect there. Assume also that the agent is able to observe these causal chains, and in particular to observe the eventual consequences of its own actions in the environment. The preferences and dispreferences that the agent already applies to the immediate effects of actions can also be applied to their indirect consequences.

When the agent addresses one specific task in the framework of its current goal structure, it may not be possible for it to identify all the possible indirect consequences of a given plan. This is in particular so in dealing with consequences that are merely possible, with a small probability of occurring, but resulting in a very bad disadvantage. It would be reasonable for the agent to let repeated observations of negative indirect effects of actions and scripts lead to a revision of its current goal structure, so that such action plans can be rejected because already their predicted direct effects are inconsistent with the agent's goal structure after the revision.

In this case as well, the benefit of the revision of the goal structure should be that it improves the system's ability to deal adequately with situations and tasks that it encounters and within the framework of its overall preferences concerning the state of its environment. It is therefore another example of self-modification of behavior.

The behaviors under consideration may be the physical actions of a robot, but they may also be behaviors of other kinds. In particular, the choice of actions that are consistent with a number of standing goals, and the modification of the standing goals based on observed negative effects may also be applicable to communication actions and to the constraints that are applied to such actions.

So far, we have only discussed autonomous self-modification that arises as a result of the agent's own experience, and in particular by accumulating methods or scripts that have turned out to be successful. However, self-modification in people arises often as a result of *interactions* with other people, where a person becomes convinced of the need for changing some aspect of his or her behavior. This would be a very interesting facility in an intelligent autonomous agent as well.

To what extent can the architecture that has been described in the previous sections of the present report support self-modification of the two kinds that have now been proposed, i.e., evolution of scripts and of goal structures? We have already remarked that it can harbor the information structures that are used by Soar, which provides one datapoint in this respect. More generally, we notice that the representation framework has the ability to express both scripts, records of past action sequences, and expressions in logic that can be used e.g. for expressing task-specific as well as standing goals.

Furthermore, the architecture provides systematic ways of preserving older policies, in particular using the version management system. The Soar example provides a relatively weak case, for as long as "chunks" are accumulated but never removed there is in fact no need for version management. Of course, at some point the unused information has to be removed from actual use, and then archiving may be of interest.

There are however stronger examples. Autonomous agent systems in some contemporary robotic projects (examples forthcoming) perform considerable processing in the elaboration phase of their decision procedure, to use the Soar terminology, and this typically results in a plan that can be quite large, up to several thousand lines of conditions and actions. It is clear that plans of this size, or major building-blocks that are used in such plans are likely to evolve at a fairly regular pace. Also, systems of this kind are likely to generate a number of plans that are similar but not identifical, for addressing situations that are similar but not identical. Version management is then likely to be very useful as a basic resource in the system.

The use of goal revision is also a case in point, since the set of standing goals for an autonomous system can not be viewed as a collection of independent goal elements; it should be seen as an integrated structure. This means that also in this case there is a need for archiving an older version of the goal structure after that structure has been modified.

On the other hand, it is also clear that self-modification functionality of the kind described here will require additional facilities, in particular of a computational character. This includes algorithms for planning and plan revision, for causal and diagnostic reasoning, and for reasoning about goals and the consequences of adopting particular goals. From an architectural point of view, the important question is not whether these facilities exist already or not; the important question is whether it is appropriate to implement them as a higher layer on top of what has already been defined and implemented, or whether they should instead have been made an integrated part of the kernel. The reason for the latter alternative would be that some of the additional facilities could simplify the tasks or the structure of some existing part of the system kernel. We do not see any such possibility, and therefore propose that the Leonardo architecture described in this report is suitable as a first layer in the implementation of a system that is capable of (among other things) autonomous self-modification.

8 Discussion and Conclusions

The term "self-modifying program" has negative connotations in software technology. The established wisdom is that self-modifying programs are useless toys, except for viruses and other software pests that use self-modification, and that in any case they are impossibly difficult to document and to maintain. This view was established in the early days of computing and has persisted since then.

It is a paradoxical position, however, since software is written in order to be executed in a von Neumann-type computer, and *the* most characteristic and unique feature of the von Neumann computer is that it can modify its own programs. Why is it that contemporary software technology does not allow this unique capacity to be used to its full power?

The truth is, of course, that self-modification *is* used in practice, although we do not think of it as such. Every time that a new piece of software is installed on a computer, and every time that a software update is received and installed, and every time that a user changes the dynamic definition of a field in her spreadsheet, the effect is that "the program" of that computer has changed. However, these self-modification operations are often mystified, for example by using the term "wizard" for the program that executes an installation script.

It is our thesis that self-modification is a legitimate topic in software technology. Practical, controlled self-modification should be related to autonomous self-modification which is an important research area. The entire topic of self-modification merits a systematic study of the design principles that are needed in order to support it in a clear and comprehensive way. The first of these design principles, in our view, is that the entire system must be organized in terms of an information structure consisting of entities and information that is associated with these entities. The term "self-modification of software" should not be identified with the autonomous modification of a sequence of machine instructions, or a sequence of lines in a conventional programming language. A software system should not be seen as a structure of compiled modules that can invoke each other during execution; it should instead be seen as an information structure that has a large number of program fragments and scripts attached to it. Self-modification of the "program" should then be understood as the autonomous modification of that richly connected structure.

The architectural philosophy of the Soar system is a well-known early example of this view of software systems, and A.I. systems are in fact often built in this way. I propose however that the same philosophy is well motivated for many kinds of practical software systems as well, and that it should be considered as a standard method in practical software engineering.

A second design principle is that the information structure that holds a software system together must be expressive enough. The question of expressivity is however a difficult one to analyze, since in principle any digital representation of information can represent any other in some way. The question is therefore what are the consequences of a particular representation when one constructs a knowledgebase, and when one is to define scripts or rules.

For example, one difference between the representation of states in Soar and information states in Leonardo is that Soar does not work with composite expressions. Therefore it is necessary to introduce additional 'objects' in order to construct e.g. a set or a record. This is certainly always possible, but we propose that it makes the rules much more complicated, and that manipulation of the rules therefore becomes more difficult. The same information structure as in Soar is also used in representation languages such as RDF and OWL.

The self-description in the Leonardo system makes extensive use of composite entities and structured attribute values, and provides ample evidence for this design principle. In addition, our discussion of autonomous self-modification indicated that such a facilities will place even stronger demands on the representation system. For example, both the Prolog language and more general theorem provers for first-order logic support the use of functions in the sense of first-order logic, and using them purely relationally is a very inconvenient restriction.

A third requirement on the information structure is that it must be understandable for the human, which includes the requirement that its textual representation must be easily readable. In fact, the same readability considerations that we apply to programming languages should also be applied to the textual representations of information structures. This immediately disqualifies XML as a notation in this context, regardless of what advantages it may have for other purposes.

Our approach to the topic of self-description and self-modification is to work bottom-up. The present report has described relatively mundane self-description and self-modification facilities. Still, one observation that one can make for the design described here is that it contains many interdependencies. It is not just a collection of independent facilities; its different parts depend on each other so that the whole is more than the sum of its parts. According to this experience it is worthwhile to give full attention to the basic layers of the self-description architecture before proceeding to higher-level facilities, so that the synergy effects between different parts of the basic layer become as strong as possible.

One advantage with this bottom-up approach is that it is realistic to implement the basic layers in moderate time and to obtain extensive experience with it. By the way, there exists of course a working implementation of the design that has been described in this report, with merely minor exceptions.

Although the present design is a software tool in its own right for existing and forthcoming, practical applications, it is also intended as a platform for more advanced services, in particular for autonomous action. With respect to artificial intelligence, we propose that in order to construct *an intelligent system*, and not merely *computer programs that use some A.I. techniques* the most viable strategy is to build such systems on a platform that provides self-description and self-modification facilities in a systematic way and beginning with a number of very practical aspects of software operation. The Leonardo system whose self-description and self-manipulation facilities have been described here is proposed as a candidate for that platform.