KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Managing Information Aggregates in the Knowledge Representation Framework

Chapter 1

Command Files and Representation Files

The Lisp language has a notion of *properties* whereby each combination of two symbols, called the *carrier* and the *property* can be assigned a corresponding *value.* The value can be an arbitrary S-expression. Properties are used for storing information during a session so that it can be retrieved and used later on during the same session. The Knowledge Representation Framework uses the same design, but distinguishes between *attributes* and properties: the values of attributes can be arbitrary KR expressions, whereas the values of properties in KRF must be strings. Usually they are used for "long strings" consisting of several lines of text.

Property values and attribute values are retained as datastructures within each session with a processor, but they will be lost when the session terminates unless there are some special arrangements for saving their values and for recovering them during later sessions. The methods for doing this were already discussed in the KRF Overview report, but here we shall address the same topic with more detail and in the context of an actual implementation.

The most straightforward way of preserving the values of attributes and properties is using ordinary (text-formatted) files containing the textual representation of these values. An alternative possibility is to preserve the entire session using a memory dump. If the former choice is made then again one possibility is to leave it to the user to implement routines that construct appropriate property assignments at the beginning of a session, and that save relevant parts of them at the end of a session. However, the Interlisp system pioneered a method of *generic save and reload* where there are general-purpose routines that save and reload property values. These routines rely on catalogs that define what information is to be preserved on what files. These catalogs are present within each session; they direct the data saving process and they are also saved on the preserving files, so that they can readily be obtained when those files are reloaded during a later session. In short, the catalogs of the preserved information are themselves part of the preserved information.

Any save and reload scheme makes it possible for the user, in principle at least, to use two different methods for extending and modifying the information in the system. It is possible to do such changes in interactions with the system during a session, relying on the fact that the changes will be preserved during the next save operation. It is also possible to text-edit the saved files and to reload them into the session, even in the course of a session, thereby overwriting the results of previous loads of the same file. The practical convenience when using these methods depends of course on the quality of the interaction during sessions with the system, and on the readability and convenience of the textual representation.

The method of generic save and reload has been retained and developed further in the Leonardo system, and it will be described and discussed in the present chapter.

1.1 Command Files

The `put` command has three arguments - a carrier, a property and a value - and can be used for assigning a value to a combination of a carrier symbol and a property symbol. It works in the same way in Lisp and in Leonardo, as follows in the Leonardo CLE loop:

```
put Bjorn has-children <Karin Sven>
```

and as follows in `Lisp`:

```
(put 'Bjorn 'has-children '(Karin Sven))
```

The value can be accessed using the function `get,` as follows in the two languages:

```
(get Bjorn has-children)   =>   <Karin Sven>
(get 'Bjorn 'has-children)   =>   (Karin Sven)
```

In the Leonardo case, the `put` command-verb and the `get` function can be used both for attributes and for properties.

In the method of *command files,* property values are preserved using files that contain a sequence of commands, including for example `put` commands. The saving routine generates such files, and the reloading of the information during a later session is done by reading successive commands from the file and executing them one at a time. The file contents need not be restricted to using the command-verb `put,` of course. For example, if the relation of being married is represented as properties both ways between the husband and the wife, then it may be convenient to have a function that takes the identifiers for the two persons as arguments, and that assigns both properties (or attributes), and then one may choose to let the information-preserving file contain a number of commands of that kind.

1.2 Representation Files

A *representation file* is a file containing a (usually) textual representation of some information. KRF entityfiles, which were described in the KRF Overview are examples of representation files. The major advantage of representation files is that they may be more readable than command files, since the latter are restricted to the notation used for the commands. Compare the following simple examples of the two approaches, first a part of a representation file:

```
----------------------------------------------------------------
-- red

[: type color]
[: has-examples {rose ruby poppy}]
[: translations {[: french rouge][: german rot]}]

@Associations
The red color is associated with a lot of symbolism, such as
denoting "stop" at traffic lights, representing particular
political movements, and being recognized as the color of love.

----------------------------------------------------------------
-- blue

[: type color]
[: has-examples {bluebell forgetmenot}]
[: translations {[: french bleu][: german blau]}]

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

and then a part of a command file:

```
[put red type color]
[put red has-examples {rose ruby poppy}]
[put red translations {[: french rouge][: german rot]}]
[put red Associations
"The red color is associated with a lot of symbolism, such as
denoting "stop" at traffic lights, representing particular
political movements, and being recognized as the color of love.
"]
[put blue type color]
[put blue has-examples {bluebell forgetmenot}]
[put blue translations {[: french bleu][: german blau]}]
```

This example shows how, in a command file, it may be more difficult to get an overview of the information contents and to distinguish its various parts. This problem may be mitigated by the judicious use of whitespace (i.e. space characters and blank lines) and comment lines, but the approach of the representation file gives by definition more opportunities for readability-improving measures.

If the intention is that information contributions are mostly to be made using commands or other interactions during sessions with the system then the readability of saved files is unimportant. This may be the case when all information has very simple structure. However, when more complex information is used, an in particular if program modules and scripts are also to be considered as part of the same information base, then it is usually a requirement that one shall be able to edit the textfile representations of information and load them into the session, and then the readability of these files is an important consideration, and there is a case for using representation files.

1.3 Embedding Command Sequences in Representation Files

The strict use of the representation-file approach has the disadvantage of being somewhat restrictive, and there are cases where one would like to use representation files for the major part of the information, but to also have an option of using commands in the input files in those situations where this is more convenient. Although one may of course use an approach where some of the files are command files and others are representation files, it is more convenient to use representation files where some of the attributes or properties contain commands, and to define the loading process so that it executes commands when it encounters them in specific positions of a representation file that is being loaded.

The Knowledge Representation Framework uses representation files where attributes and properties are displayed in different ways, as shown in the above example. The values of properties are in fact multi-line strings, and one use of property-values is to contain a command or a sequence of commands. This combines the advantages of the two approaches.

Incremental programming systems, such as those for the `Lisp`, `Perl` and `Python` languages, contain a built-in command that loads a given file in the sense of reading it and executing the commands in it one after the other. Data files using the command-file approach can therefore be read directly in such systems. Data files using the representation-file approach must instead be read using a separate data parser which may slow down the operation. If this gives rise to a performance problem then it can be solved by using a *dual* approach where there are two save operations, one for saving a representation file and another one for saving a command file, as well as the corresponding load operations. Particularly large files are saved and reloaded as command files; changes to their contents are made via the command-line executive; but one retains the possibility of also writing, editing, and reloading the corresponding representation file in those situations where this is worth the resulting time delay.

Conventional programming systems for languages such as C++ do not offer these possibilities, and there it is necessary to use a separate parser regardless of whether the command-file approach or the representation-file approach is being used.

1.4 Representation Files in the Leonardo System

The Leonardo system is an implementation of the Knowledge Representation Framework and is presently written in Allegro CommonLisp. It uses the representation-file approach and allows commands to be embedded in the values of certain specific properties of the representation files. The property

`Execute/Lisp` is used for commands that are to be executed by the underlying Lisp system, and the `Execute/SCL` property is used for commands that shall be executed in the same way as if they were typed to the command-line executive. The following is a trivial example of an entity-description using these facilities.

```
---------------------------------------------------------------
-- abs

[: type leofunction]

@Comment
Obtains the absolute value of the number given as argument.

@Execute/Lisp
(defun abs (a)(if (< a 0) (- a) a))

@Execute/SCL
[def [equal (abs .a) (if [ls .a 0] (- .a) .a)]]

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

SCL commands appear with their proper syntax, which means that they are enclosed in square brackets. These square brackets are optional in command-line input, but they shall be present in all other contexts.

Chapter 2

Operations on Attributes and Properties

The following operations apply to attributes and properties in the Leonardo system. A larger reportoire of Leonardo system commands are documented in the technical report "Facilities in Leonardo."

2.1 Command Verbs

`[put .c .a .v]`

The first two arguments must be entities; the third argument can be any KR expression. The effect of executing this command is that subsequently the term `(get .c .a)` obtains the value `.v`

2.2 Functions and Predicates

`(get` ***c a***`)`

Both arguments must be entities. The value is the most recently assigned value according to the `put` command or according to an entityfile that has been loaded using the `loadfil` command or equivalent. If no value has been assigned then the function returns the symbol `nil.`

`[has-location` ***ef***`]`

This literal is true iff the argument is an entity that serves as the name of an entityfile. The concrete condition is that ***ef*** is associated with information specifying the location of the entityfile in the directory structure at hand. Ω

`[exists-file` ***ef***`]`

The literal is true iff the entityfile ***ef*** is associated with information about where the corresponding file is to be located in the directory structure, and this file actually exists. Ω

2.3 Saving Assigned Values

Leonardo distinguishes between *attributes* and *properties,* although the operations `get` and `put` as well as other similar operations can be used for both attributes and properties. The differences are as follows.

- Attribute values shall be KR expressions and are represented accordingly as data structures in computational sessions. Property values are considered as strings, and often they are multi-line strings.
- When an entityfile is written using the `writefil` command or equivalent, then the type of an entity determines which attribute values are going to be written. The type of the entity, as specified by its `type` attribute, shall be again an entity having an `attributes` attribute whose value is a set or sequence of entities indicating which attributes are to be written. – For properties there is no such rule, and instead the `writefil` operation will in principle write back the properties that were in the entityfile when it was loaded (more about this below).
- In order to help with the distinction, there is a recommendation that attributes shall be written with a small first letter and properties with a capital first letter. However this is not a binding rule.

With respect to properties, the actual rule is that each entity has a hidden attribute (not mentioned in the `attributes` attribute, and not written into the textual entityfile) called `textprops` whose value shall be a set of the symbols that occur as properties for the entity in question. The value of this attribute is set when a entityfile is loaded and for the entities that occur in that entityfile. Therefore, if one should wish to add or remove properties during a session, this is done by changing the value of the entity's `textprops` attribute.

There are a few reserved attributes that need not be included in the list of attributes in the `attributes` attribute, and that are written into the result file anyway. This includes in particular:

- `type,` showing the type of the entity
- `in-categories,` where the value shall be a set of entities that indicate particular characteristics of the entity in question
- `latest-rearchived,` which is used by the built-in version management system in Leonardo.

Chapter 3

Operations on Entityfiles and Knowledgeblocks

3.1 Knowledgeblocks

The following operations are used.

```
[crek .kb]
```

Creates a knowledgeblock with the name specified in the argument.

```
[setk .kb]
```

Selects the (existing) knowledgebase `.kb` as the current knowledgebase, for use in some of the following commands.

The names of knowledgeblocks should be selected so as to end with the three letters `-kb` – if this does not happen then the last three letters of the name will be removed anyway. Introduction of a knowledgeblock called for example `colors-kb` will result in the creation of a subdirectory called `Colors` for the agent in question, and the creation of a *catalog file* called `colors-kb` in that directory. It will also lead to the creation of a sub-subdirectory `Colors/cl` which is used for command files expressed in CommonLisp, using the dual file approach that was described in Section 1.3.

The role of the catalog file is to contain information about which additional entityfiles are created in the knowledgeblock in question, and also to provide a place for attached procedures that are to be invoked each time the knowledgeblock is loaded. The locations of the entityfiles in the knowledgeblock are usually in the directory that was created for that knowledgeblock, i.e. `Colors/` in the example, but there are a number of cases where one wishes to locate them elsewhere. Please take a look at the contents of such a catalog file and you will see how it is organized. (You will see that there is a certain redundancy in the attributes in a catalog file, but this is for legacy reasons in the system).

The location of the catalog file can not be in the directory of the knowledgeblock since this would lead to a circularity in the access mechanism. It is instead specified in a meta-catalog that is located in

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

as seen from the agent's top-level directory. This entityfile is therefore updated when a new knowledgeblock is introduced.

The entity representing a knowledgeblock has a number of attributes that control how the knowledgeblock is loaded, in particular the attributes `requires` and `mustload.` The value of `requires` shall be a set or sequence of other knowledgeblock entities, and the value of `mustload` shall be a set or sequence of entityfile entities. Both attributes are optional. The `loadk` command is defined so that it will first load the entityfile given as its argument, then make a loop over the elements of its `requires` attribute and apply the `loadk` operation on them, recursively, and finally apply the `loadfil` operation on the value of the `mustload` attribute. This feature makes it possible to declare how some knowledgeblocks depend on some others.

For some odd reason that has not yet been understood it does not work to include `indivmap-kb` in the `requires` attribute and it has to be loaded by a separate command by the user.

The value of the `mustload` attribute will usually be entityfiles in the same knowledgeblock, but this is not necessary and it may also contain entityfiles from other knowledgeblocks, as long as their respective catalog files have been loaded first.

3.2 Entityfiles

The following commands are used.

`[crefil .ef]`

Initializes an entityfile called `.ef` in the current knowledgebase.

`[loadfil .ef]`

Loads the entityfile `.ef` into the current session.

`[writefil .ef]`

Rewrites the entityfile `.ef` in its place in the directory structure, using the catalog of the *entity* `.ef` that is present in the current session, and the current values of the attributes and properties of the entities that are in that catalog.

`[curfil .ef]`

Sets the entityfile called `.ef` to be the current entityfile, as used in the following commands.

`[addent .e]`

Adds the argument `.e` to the contents of the current entityfile. Notice that in order to be included when that entityfile is written, the entity `.e` must have a value for the attribute `type`, and a value for at least one attribute or one property.

`[loadf]`

Like `loadfil`, but for the current entityfile.

`[writef .ef]`

Like `writefil`, but for the current entityfile.

The following specialized functions may be useful.

`(filemembers` ***ef ty***`)`

Obtains the set of those members of the entityfile ***ef*** whose type is ***ty.*** Type is only defined directly as the value of the `type` attribute and does not allow for subsumption from a supertype. Ω

Each entityfile is represented by an entity having the same name as the entityfile. Thus if the knowledgblock `colors-kb` contains an entityfile called `warmcolors` then it will (usually) be located as `Colors/warmcolors.leo` and the entity `warmcolors` will be the first entity in the entityfile. Its most important attribute will be `contents` where the value shall be a sequence (or set) of entities, with itself as the first member.

The value of the `contents` attribute is used for specifying which entities are going to be written to a file under the `writefil` operation. The `loadfil` operation reconstructs this value from the actual contents of the file that it reads, so there is no need to modify the `contents` attribute yourself when you text-edit an entityfile.

There is a special (and somewhat dangerous) rule saying that when the `writefil` operation writes a result file, it will omit entities that do not have a value for the `type` attribute, and it will also omit entities that, although they do have a value for `type,` they have neither any property assigned, nor any value for any other attribute besides `type.` This means in particular that if you happen to mis-spell the value of the `type` attribute then the entity will be omitted. As long as you are a novice to the system it is therefore strongly advised to make backup copies of entityfiles before you do `writefil` operations.

The list of entities of the entityfile `.ef` is obtained as the value of `(get .ef contents).` While working with a system one may sometimes wish to find out what was the entityfile where a particular entity `.e` is defined. This can be obtained as `(get .e read-in-file).` The hidden attribute `read-in-file` is assigned when an entityfile is loaded.

Chapter 4

Episodes and Microworlds

The purpose of the episode construct in Leonardo is to set up a framework for representing past actions and possible future actions. This is important for several purposes, including for planning, for anticipation of the future as influenced by other agents, for 'diagnosis' of the causes of past or current situations, and for case-based operations.

The purpose of the microworld construct is to make it possible for the system to restrict attention to a limited number of entities, in particular, entitites that represent things in the system's environment, in order to make it possible to analyze these things and their relationships in sufficient detail.

Leonardo has chosen a realization of these two constructs whereby they are tightly integrated, so that each episode is associated with a set of entities constituting its microworld.

4.1 Episodes – General operations

The following representation is used. There is a type for episodes, and individual members of that type are normally called `episode-0000, episode-0001,` and so on. The starting procedure for a Leonardo session creates `episode-0000` as an initial episode and assigns attributes to it. It is not included in any entityfile, so if the user wishes this episode to be preserved after the end of the session, it is up to the user to include it in a suitable entityfile.

At each point in time during a session there is a *current episode* for it. Actions that are executed, for example as the result of command input, are added to the history-list of the current episode. Additional episodes can be created using commands. These can either be *sub-episodes* meaning that they are included into the episode where they are created, or *side-episodes* meaning that the episode starting them is ended when the new one takes over, and they are both sub-episodes of the same super-episode.

The following operations are used for episodes.

`[Cur]`

Create a new episode that is directly subordinate to the top episode, `episode-0000,` and make it the current episode.

`[curep]`

Display the past actions in the current episode, in reverse order.

`[topep]`

Display the past actions in the top episode, in reverse order, and then show the list of its sub-episodes. These may have been created using the `Cur` command.

4.2 Microworlds

The facilities for microworlds are not part of the general Leonardo system, and are made available in library form. The remainder of this chapter will describe how it is defined for the purpose of the AI course TDDC65.

A *microworld* is in this case as an entity that can generate one or more episodes, and that is associated with a particular set of verbs, namely, verbs for actions that can be executed in those episodes. A *microworld episode* is an episode where actions defined by a particular microworld can be performed. Each microworld episode is associated with a set of *included entities,* that is, entities that the microworld's actions can have as arguments in that episode.

The sequence of actions in the microworld is therefore a sequence of actions that operate on a well-defined set of objects and with a selected set of verbs. The session where the microworld session executes normally contains many other entities and many other actions, for example system actions such as loading and writing entityfiles.

When microworld episodes are used e.g. for planning and for case-based learning, it is normally the case that many episodes, and often quite small ones are generated for the same microworld. An additional use of microworlds arises in labs for our course, since labs are set up so that they are completed when the student has been able to construct a microworld episode with desired properties. When he or she has achieved this then the successful episode is placed in the lab report file which is uploaded to the registration agent for approval.