Intelligent Software Individuals Based on the Leonardo System

Erik Sandewall
Department of Computer and Information Science
Linköping University
S-58183 Linköping, Sweden

Abstract

This article proposes a suite of design decisions for the overall design of an Artificial Intelligence, i.e., a software system that exhibits intelligence in the spirit of the early days of A.I. research. The key aspects of the proposal are: (1) The identification of the A.I. system as a *software individual* that has the properties of integrity and persistence; (2) The construction of a *software platform* that integrates aspects of incremental programming languages and systems as well as of operating systems, with aspects that are intrinsic to knowledge-based artificial intelligence; (3) The use of a *representation language* that builds on essential aspects of S-expressions, Lisp, logic and extended set theory, but which is used both as a vehicle for software and as a publication language e.g. in lecture notes; (4) The identification of *actions* and aggregates of actions as first-class citizens in the representation language and as an important type of data object in the software system.

The article also describes the Leonardo software platform, its representation language, its educational resources and its knowledgebase library which is one implementation of these proposed design decisions. Finally it makes a proposal concerning the research paradigm for this research area.

The design of a software system that *has intelligence* in a sense that is commensurate with how humans have that property, is only possible if there is a coherent design for the *software architecture* of that system and a strategy for the *knowledge acquisition process* whereby it can build and extend its knowledgebase. The present article describes the design principles for the `Leonardo` software system and for its means of communication and knowledge acquisition. It concludes with a proposal concerning the research paradigm for research towards software that has intelligence.

Rather than describing the Leonardo system directly, I shall first identify a sequence of *design decisions* which together characterize the system. This is not a historical account of the design decisions that were actually taken in the earliest stages of the work that led up to the current Leonardo system; several of them are posterior reconstructions. They are related to the actual design history in the same way as a polished mathematical proof is related to the discovery process for that proof.

The design decisions will be specified as three successive *layers* where design choices in earlier layers are necessary for choices in later layers to make sense.

Layer 1. Software Individuals

The abstract design process towards a software artifact that exhibits intelligence in the original sense of Artificial Intelligence research, begins with the following design decision.

Decision 1. The software artifact must be organized as an entity that consists of programs (i.e. executable computer code) and data and that has the following characteristics:

- 1.1. It has integrity and persistence, so that it can exist over a considerable period of time and be distinguished from other similar artifacts.
- 1.2. During the period of its existence, besides a capability of interacting with its environment, it also has the capability of learning, both in the sense of acquiring knowledge, and in the sense of modifying its contents in response to its experience.

I shall use the term *intelligent software individual* for such an artifact.

The importance of Decision 1 is that it sets a specific goal: the construction of a software individual with certain properties as described in the later decisions. For example, if a software individual is manifested simply as a directory in the computer file system with its subdirectories and the files in them, then it has the following elementary properties: If you make a copy of it, then you obtain a new individual. If you move it, for example in the sense of moving it from a hard disk to a USB stick, then it is still the same individual. Each individual may evolve over time, as described in item 1.2, which means that if you make a copy Y of an individual X and allow both X and Y to evolve, then after some time they will no longer be equal.

One important reason for adopting this view is that thereby one obtains concrete operational meanings for two important phenomena: *learning* as performed by an intelligent system, and *exchange of information* between two or more such systems. In order for learning to be of any use in an A.I. system, it is necessary for that system to have persistence for a reasonable period of time, so that successively learnt knowledge items can accumulate and be used in new situations. Likewise, exchange of information is by definition only possible if you have several intelligent systems that can engage in such exchange, and that are sufficiently dissimilar from each other so that information exchange is meaningful and nontrivial.

Moreover, one important aspect of learning and exchange of information (as well as for delegation of tasks) is maintaining a model of the competence and the credibility of other agents, and using that information in forthcoming exchanges. This is only possible if those other agents have the character of individuals, so that they can be assigned names, and so that each individual can maintain knowledge about other individuals and reason about their properties.

Learning and interaction are fundamental capabilities for intelligence as we know it in humans, and it is difficult to make sense of what an artificially intelligent system should be like if it did not have these capabilities.

Layer 2. Representation Language

The Leonardo system has been designed according to Decision 1. In the next step the following design decisions are made.

Decision 2. An intelligent software individual must be able to maintain a model (i.e. a formal description) of its own internal structure, and to analyze and modify this model.

This is not merely a requirement about the existence of a software facility for self-modification; it is as much a requirement that the individual's software shall be *organized in such a way* that it facilitates the implementation of the self-modification facility.

Decision 3. The implementation of an intelligent software individual should be based on a coherent *representation language* with at least the following capabilities:

- 3.1. It shall permit the description of both the state of some world at the current time, previous and future states in that world, and actions and processes that take place in that world.
- 3.2. It shall be possible to use it both for characterizing states and processes in the real world, and states and processes within the computer, including the software individual itself.
- 3.3. With respect to the use of logic, it shall at least subsume first-order predicate calculus and allow its use for a variety of purposes, including the condition part of a conditional command expression, and the precondition and the postcondition of a command verb.
- 3.4. Besides for general-purpose software development and knowledge representation, this language shall also be a viable platform for the definition of special-purpose languages and of the configuration languages for a variety of software engines.

Decision 4. The representation language described above should be a convenient notation to use for humans, and in particular, for the persons that design the various parts of the intelligent software individual. A concrete test for this is that it shall be attractive to use it as the main formalism in textbooks and research articles.

One reason for adopting Decision 3 is that otherwise it will be difficult to implement Decision 2. Subdecision 3.1 is obvious, but it is identified in order to provide the context of the following subdecisions.

Subdecision 3.2 is about applying the same representation techniques to states and processes within the software system as are used for modeling the system's external world. Concretely speaking, this means for example that if an intelligent software individual is embedded in a mobile robot, then the representation language for physical actions of this robot shall also be used for shell-language-type commands to the operating system that is used in the robot and that serves as the host of that intelligent software individual. As another example, a software update operation in that host shall also be represented as an action in that same representation language.

An immediate reason for such a design is that the capability of self-modification ought to be a basic characteristic of A.I. software. Self-modification of software should not only be understood as a capability of replacing individual lines of code or segments of code in a large, conventional program; it may as well be understood as a capability of reconfiguring the modules in a large, modular system, adding or removing modules there, and changing the reportoire of rules, scripts, or axioms that are expressed in an appropriate logic.

However, Subdecision 3.2 can also be understood from a software-oriented point of view. While the use of preconditions and postconditions of actions is basic for action planning in A.I., preconditions and postconditions of segments of code is also basic in formal approaches to software engineering. The only difference is that software engineering usually takes a verification-oriented view, so that preconditions and postconditions are used when code is validated and compiled, whereas action planning in A.I. is usually used dynamically i.e. "at run time." This difference is not a major one, and a unified treatment of computational and physical actions would seem to be both natural and promising.

As another example, the structure of tasks and subtasks in a Hierarchical Task Network (HTN) is analogous to the structure of process invocations in systems software, but it differs from the latter by being oriented towards interpretive use.

One may also see Subdecision 3.2 in a historical perspective. It used to be that Artificial Intelligence developed and used its own software technology based on the Lisp programming language, Lisp machines that integrated the operating system and the programming language, and a family of S-expression-based representation languages such as KIF, KQML and PDDL. This era is gone, and the contemporary wisdom is that we shall use conventional operating systems and programming languages (such as Java, or Python), and conventional database systems.

A consistent implementation of Subdecision 3.2 may bring us to a new situation that combines aspects of the earlier and the present era, namely, to a software technology where the basic software layers in the computer (i.e. those performing the services of an operating system and a programming language) are organized in a way that realizes some of the needs of an A.I. system. In particular, the explicit representation of actions, their preconditions and effects, and software for making use of that information would be part of the basic software layers, and they would be realized in such a way that they are immediately useful for A.I. purposes. However, unlike the situation in the earlier era, this should not be a software technology that is used only by A.I.; it should be realized in such a way that this kind of basic software is attractive for use in all major kinds of computing.

In other words, instead of using a software platform consisting of conventional operating systems, programming languages, database systems etc., and having to build A.I. software on top of these software layers, Artificial Intelligence would profit very much if some of its essential software techniques could dissipate into those underlying layers. Conversely, this could also be an important contribution to software technology in general.

Subdecision 3.3 concerns the use of logic, but in particular it urges a return to the use of first-order predicate calculus (FOPC) as a lingua franca for knowledge representation. There exists already a small number of well-established predicates and functions, such as the *Holds* and *Do* predicates, the successor function in situation calculus, and so forth, and it would be natural to extend this reportoire gradually. Current research publications in A.I. do not contribute to such a development, however, and quite possibly because contributions of that kind are not well received. When knowledge of a particular kind is to be represented in logic there is a choice between introducing a new, specialized logic with its own inference rules and its own semantics or, alternatively, extending a currently used first-order theory with appropriate predicates, functions, and applicable axioms. The former approach provides much better opportunities for formal analysis and for finding "interesting" metalevel theorems, whereas the latter approach may easily be discounted in a journal reviewing process for not having produced any "essentially new" results of the required depth.

However, for actual construction of A.I. systems it is not an attractive prospect to have to combine a number of separate, entirely unrelated logics, and the use of a uniform logical platform is indispensable. Although there have certainly been proposals for alternatives to FOPC as such a platform, none has made a credible case. There will be a need for particular extensions, for example in the direction of nonmonotonicity, but the representation language should first of all include FOPC as a subset.

It is true that FOPC is used extensively in systems such as as CYC and in ontologies such as SUMO. However, regardless of the actual quality of their representation languages, it is unfortunately the case that our research literature in the sense of journal and conference articles neither contributes to, nor makes any use of the vocabulary of predicates and functions that is defined in these languages. One must conclude, I think, that the accumulation of a publication-quality knowledge representation vocabulary must start in the formal research publications themselves.

Subdecision 3.4 can best be understood as a decision to establish and use a good *syntactic style* for the variety of notations that will be needed in the entire A.I. system. Please think of *S-expressions* as one syntactic style, and *XML-like expressions* as another one. The important thing about a syntactic style is that it defines a generic way of writing formal expressions, within which one can define a number of specific languages. For example, S-expressions are used by both Lisp, KIF, KQML, PDDL, and several others. The XML syntactic style which is characterized by tags surrounded by angle brackets is used likewise.

In fact, the use of syntactic styles is an alternative to the use of parser generators and compiler-compilers. Parser generators encourage you to define a new syntax, for example using BNF notation, for every new language that you define. The use of a syntactic style encourages you instead to write one single parser that converts expressions in the syntactic style to an internal representation as a data structure, and to implement each of the languages that adhere to the same style as interpreters and support software that operate on that internal representation.

The use of a uniform syntactic style has many advantages, and it supports the notion in Decision 3 that the A.I. software system should have a model of its own software structure. However, the problem that we have is that neither of the two currently widespread syntactic styles (S-expressions and XML) satisfies Decision 4, namely, the requirement that expressions that are formed using this style shall be convenient to read for the human eye and mind. This applies in particular for SUMO, which uses the S-expression-based KIF notation, and even more for the CYC language.

For the Leonardo system we have therefore introduced an alternative syntactic style, called *Knowledge Representation Expressions* or KR-expressions, which is used repeatedly for a number of specific languages within the system, including the *Common Expression Language* (CEL) which is used for representing actions. KR-expressions are similar in spirit to S-expressions, and CEL can be seen as a counterpart of Lisp and of KIF, but based on KR-expressions rather than on S-expressions. KR-expressions and CEL will be further described below.

Layer 3. System Structure and Subsystems

For a system that adopts design decisions 1 through 4, such as Leonardo, the obvious next step is to implement a *software platform* that can be used as the basis for intelligent software individuals. I propose that this software platform shall integrate traditional aspects of systems software with traditional aspects of A.I. software architectures, as follows.

Decision 5.

- 5.1. The software platform provides the same facilities as incremental programming languages, such as Lisp and Python, as well as major aspects of the shell part of an operating system.
- 5.2. The platform also identifies *actions* as first-class citizens, and it is able to represent various kinds of information about actions, including the history of past actions in the system, the predicted and actual effects of actions, and others more. Execution of an action is treated differently from the side-effect-free evaluation of a term.
- 5.3. The platform can be used in a simple command-execution mode, similar to a contemporary operating-

system shell or incremental programming system. However, its executive shall also provide many of the services of A.I. software architectures, for example according to the BDI model, the SOAR model, or Hierarchical Task Networks. Moreover, even if it is used for simple command execution, it shall be able to use the information about actions according to item 5.2 in order to facilitate the execution of actions as well as error handling in case of action failure.

- 5.4. The platform makes uniform use of a syntactic style and implements a representation language within this syntactic style, along the lines of Decisions 2 and 3. Actions and the information about actions is also represented in this representation language.
- 5.5. Each instance of the software platform shall be an identifiable software individual. It shall be able to model other individuals in the same way as it models itself, and it shall also be able to model the communication network and communication procedures that allow it to interact with other individuals. Such interactions are represented as particular kind of actions.

The Leonardo System

The Leonardo system [1] is a computational platform that implements the design decisions that were described above (except for the learning aspect mentioned in item 2 of Decision 1 which is future work). The present section will describe the system as presently implemented.

Platform Structure

The Leonardo system is implemented in CommonLisp and runs on both Allegro CommonLisp and the Clisp implementation. Each copy of this platform counts as a Leonardo individual.

Leonardo individuals have two distinct manifestations: a *static manifestation* as a file directory with its subdirectories in the file system hosting the individual, and a *dynamic representation* as one or more runs of a Lisp system where essential parts of the static representation have been loaded. These manifestations are interdependent since a run of the individual will regularly update entityfiles in its static manifestation. The static manifestation is therefore what guarantees the persistence of the individual (cf. Subdecision 1.1). It can be moved to another part of the file system or to another memory device, and it can be renamed, but it is still the same individual and identifies itself as such. In other words, Leonardo individuals are *mobile agents* in the sense that they can be moved passively from one location to another one.

We have not implemented the procedure whereby a Leonardo individual would also be able to *move itself* from one location to another one. There is no known difficulty with doing this, but there has just not been any need for such a facility.

A special piece of code helps an individual to identify whether it is really itself or a clone of itself, i.e. it recognizes whether a move operation or a copy operation has been performed on the static manifestation of an individual. This facility can be overridden without much difficulty, but if it is not manipulated it helps to reinforce the notion of integrity of the software individual.

Each Leonardo individual consists of several *agents* including exactly one *kernel agent* which does not differ very much between individuals, and other agents that are specialized to particular purposes. Each run pertains to one particular agent in an individual, and messages can be exchanged between runs of different agents, either in the same individual or in different ones.

The design of the Leonardo platform, and in particular the design of the kernel agent integrates issues that are characteristic of A.I. and Knowledge Representation, with issues that are characteristic of incremental software systems in general. For example, specific attention has been given to how a Leonardo individual can recognize and model the computer network environment where it operates together with other individuals. A model of computer hosts, detachable memory devices, local area networks, port numbers, and so forth is necessary in order that an individual shall be able to send messages to another individual that it knows by its name, given that the location of the recipient in terms of an IP number may vary. This concern with conventional computation issues may seem remote from the needs of Artificial Intelligence, but in fact it is necessary in order to obtain a platform that can serve the needs of ordinary computation as well as providing basic services that have an intrinsic A.I. character.

The Leonardo Executive

The Leonardo Executive is a combination of a basic command-line executive (CLE) and a Hierarchical Task Network (HTN). The HTN part can be disabled in which case only the CLE part is operational. The CLE receives action expressions and executes them successively; each action expression consists of a *verb* and its arguments. The CLE differs from a conventional read-execute-print loop in the sense that the definition of a command verb consists of three parts: a *precondition,* an *executable definition,* and a *presentation script.* To process an action expression containing a specific verb, the CLE first evaluates the verb's precondition. If it is satisfied, it invokes the executable definition and delivers the outcome of the executable to the presentation script which specifies how to show the outcome to the user.

If the precondition is not satisfied then the CLE will merely inform the user of this and not proceed any further. However, the representation of the precondition as a separate entity makes it possible for the CLE to explain the fault to the user in a generic way, so that when one defines the executable one does not have to write code for checking preconditions and explaining the problem when they are not satisfied. If the HTN part is enabled then there are also other uses for the precondition expression.

The separation between executable definition and presentation script helps modularity, but it is particularly useful for distributed computation, where one Leonardo agent may request another agent to perform a particular task for it, so that execution takes place in the other agent and presenta-

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

tion takes place in the given agent. Other ways of using the separation are also possible, including dynamic switching between English and other languages in the dialog with the user.

The HTN part of the Leonardo Executive is the basis for several facilities, in particular, *precondition repair, concurrent tasks,* and *agenda management.* It is based on the introduction of a *logical clock* that defines successive timepoints or “ticks,” and where each action and subaction takes place during the interval between one tick and another, later one (possibly but not necessarily the next one).

A *task* is a particular kind of entity that can be associated with two action expressions: a *given action* expressing what the task was supposed to do, and an *actual action* expressing what was really done when the task was executed. An executed task is also associated with its *outcome,* namely, the expression that resulted from the executable and that was given to the presentation script of the action verb.

If the HTN part of the executive is enabled then the execution of an action expression proceeds as follows; this applies both for top-level actions and for subactions. First, the action’s precondition is evaluated. If it fails then execution is rejected. Otherwise, if the verb in the given action expression is an elementary verb then its execution is initiated. If the verb is non-elementary and therefore defined as a script consisting of other actions, then a new task entity is created for representing the execution of the action in question and is added to the set of current tasks. Task scripts allow concurrent actions and subtasks to be included.

At each timepoint according to the HTN clock there is therefore a set of ongoing elementary actions and tasks. Elementary actions may take one or more timesteps, so the executive checks each of them for whether it considers itself to have terminated. Similarly, for each of the current tasks, the executive checks its current subtasks and elementary actions for completion, and proceeds to the next steps whenever appropriate.

Methods that are characteristic of Artificial Intelligence are applicable if a precondition is violated for some subaction during the execution of a task script. Rather than just failing the enclosing action, the executive tries to remedy the situation. In principle this can be done in either of two ways: by first performing some *enabling actions* that achieve the required precondition, or by identifying a *substituting action* that it is appropriate to perform instead of the given one. The case of just giving up on the given action can be understood as selecting the null action as the substituting one.

The selection of enabling or substituting actions may be done in a preprogrammed way using *recovery rules* that are associated with the predicates that occur in preconditions, or that are obtained by simple deduction from failed preconditions. This is the method that is presently implemented in the system.

A more general, goal-based approach should observe that a well-founded selection of a substituting action must be based on information about the system’s goals, for example in the form a Belief-Desire-Intention (BDI) model. A plausible design would be as follows: First use a decision mechanism such as a decision tree to determine whether to go for enabling actions or substituting actions. In the former case, use traditional action planning in order to generate one or more plans that will achieve the failed precondition, but test each of the proposed plans for compliance with the set of preferences that are expressed e.g. using the BDI model. In the latter case, attempt to infer the underlying goal that the user or the invoking task intend to achieve using the action whose precondition failed, and find another plan for achieving this goal.

In any case, the various methods for resolving precondition failure lead to an amendment in the action script that is being executed. This is how there may be a difference between the given script and the actual script for a task entity. Notice, therefore, that all precondition repair takes place within a task, and by modifying the script that defines that task.

In particular, this explains the above statement to the effect that if the precondition of a single action fails then the action is rejected. The action is rejected, but if it is a subaction in a script then the repair takes place in that script, and the rejected action may reappear later on in the actual script. Single actions in command-line input are rephrased as a script consisting of that single action, so that they are also amenable to precondition repair.

Only the recovery-rule method is available in the present version of the Leonardo Executive, and the goal-based approach has not yet been implemented to completion. However, the architecture of the Leonardo system, and in particular the present HTN-based executive has been designed in such a way that it will be a good platform for this kind of robust handling of precondition failure.

Demo Application

The *Leonardo Zoo* is a software individual that illustrates the use of the Leonardo system and in particular the Leonardo Executive. It consists of two agents: the kernel agent and an agent that simulates daily events among the animals in a zoo, as well as the actions of a warden in this zoo. There is one composite action that is performed repeatedly, namely, the action where the warden makes a tour of all the animals under his responsibility, checks their well-being, and feeds them. In the absence of any reason to the contrary, the warden proceeds along a standard path and attends to one animal at a time. Sometimes this plan has to be modified, either due to something that the warden knew already at the beginning of the tour, or due to some observation or event during the tour.

This scenario requires two types of classificatory decisions, namely, identifying the possible problem (disease, etc.) for an animal with given symptoms, and identifying the appropriate cure or other action once the problem has been identified. These choices are made using a subsystem for decision trees and causal nets. (This subsystem is part of Leonardo’s software library).

Furthermore, events during the warden’s simulated tour of the zoo sometimes lead to a need for modifying the tourplan. In very urgent cases the warden may need to go at once to the Zoo pharmacy in order to pick up medical supplies; in other cases he may postpone this until a later point in the tour (if he is going to pass by the pharmacy later on anyway, for example). The choice between different ways of dealing with the upcoming needs is in turn a matter of choice based on available parameters of the current situation.

Finally, of course, the observations and the actions that are performed during one tour of the zoo are relevant for the decisions on subsequent tours, for example, if it was observed that a particular treatment of a disease did not proceed as expected. The knowledge about the expected healing or recovery process is in turn also expressed in the system's representation language.

Current Use

A number of Leonardo individuals are presently used for a variety of purposes, such as the following:

- Individuals that assist in the authoring of research articles, reports, and research-related webpages. (The present report has been prepared using such an individual, for example).
- Individuals that maintain knowledge about the restrictions that different publishers and journals impose on authors' rights to place their articles on their own webpage.
- Individuals for registration and software update services for new Leonardo individuals.
- The Leonardo system has been used in an undergraduate A.I. course in our university, where each course participant obtained his or her Leonardo individual and used it for doing the course assignments. Distribution of assignments and return of the solutions was done using message-passing between software individuals serving the students and an individual that was dedicated to course management.

Continued Development

Concurrently with its current use in applied situations we continue the development of the basic Leonardo system. The following additional design decision is intended to be used in that work.

Decision 6. The current Artificial Intelligence literature identifies several kinds of software "engines" for specific purposes, such as diagnostic engines and dialog engines. The platform that is described in Decision 5 should be organized in such a way that it subumes these engines, in the sense that some of its general-purpose executives can perform the functions of the executive in the engine in question and the remaining functionality of the engine can be obtained by plug-ins in the platform.

The Leonardo Project

The *Leonardo system* has now been described. However, the *Leonardo Project* is not merely about implementing that system; it consists of five major parts:

- Definition of a representation language along the lines of Decisions 3 and 4, consisting of a new syntactic style, a main representation language within this style, and a variety of specialized languages that also conform to the same style.
- Implementation of a software platform, viz. the Leonardo system, as described above.
- Implementation of a library of software resources based on the platform.
- Development of an Open Educational Resource (on-line lecture notes and other materials) that use the chosen representation language, and that makes use of the platform and the software resources.
- Development of an open, large knowledgebase that has a modular organization and that uses the same representation language as the Leonardo system. It is called the Common Knowledge Library (CKL).

The present section will describe item 1, and briefly describe item 4. For the CKL, please refer to the CKL website at [[2]] and for the full information about the entire Leonardo Project, please refer to the CAISOR website at [[3]]

Representation Language

The representation language is defined in three steps: syntactic style, common expression language, and entityfile syntax.

Syntactic Style: KR-expressions As an alternative to the syntactic style of S-expressions I define and use a syntactic style called *KR-expressions* which are formed recursively as described by the following scetchy overview.

Elements may be symbols, strings, or numbers as usual. Strings are enclosed in double quotes. Two specialized kinds of symbols are distinguished, namely *tags* that have a colon as their first character, and *variables* that begin with a full stop character.

An *entity* can be either a symbol (but not a variable or tag) or a *composite entity.* Composite entities are formed using a composer followed by an appropriate number of arguments, and enclosed in ordinary parentheses. Composite entities are like Herbrand expressions, or like terms in Prolog: they are only equal if identical.

Each entity has a *type* that is also an entity. Each entity may also have a number of *attributes* with corresponding *values.* For each type it is specified which attributes may be used by entities having this type.

A *set* is (represented as) an expression that is enclosed in curly brackets and that contains the members of the set, separated by whitespace. A *sequence* is a similar expression that is enclosed by a less-than and a greater-than character, serving as angle brackets on the computer keyboard. Sets and sequences are used as usual in extended set theory notation.

The following is an example of a *record:*

```
[move-to john place-4 :using car-12]
```

In general, a record is constructed using a *record former,* a fixed number of *arguments* for that record former, and optional *parameters* consisting of a tag and a corresponding value.

Finally, a *form* consists of a *function symbol* and its arguments, enclosed in round parentheses.

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

[3] http://www.ida.liu.se/ext/caisor/

In almost all cases, arguments and parameter values can be arbitrary KR-expressions, including variables but excluding tags. For details, please refer to the system documentation.

Common Expression Language The primary use of KR-expressions is for the *Common Expression Language,* CKL, which is the main representation language. It instantiates KR-expressions in the following ways (many details omitted).

- Actions are expressed as records, with the verb as record-former.
- Literals are also expressed as records, with the predicate as record-former, and without the use of parameters.
- Composite propositions are expressed as forms.
- Arguments in actions and literals are entities, numbers, strings, or aggregates of these that are formed using sets, sequences or actions.

All KR-expressions can be *evaluated* although in several cases they evaluate to themselves. In particular, all entities, strings and numbers evaluate to themselves. Sets, sequences, and actions evaluate by elementwise evaluation of their components. Logic expressions evaluate in the standard ways. Forms are evaluated in a way that is defined specifically for each function symbol. Variables evaluate to their current value in the context at hand.

In the particular case of actions (as a kind of record expressions) there is both *evaluation* and *execution.* The role of evaluation is to replace variables by their values, and to evaluate forms, but without executing the action. The separate execution operation is the one that invokes the procedure or script that is associated with the verb leading the action expression.

The total system also contains several other languages that use KR-expression style, for example a Document Scripting Language that compiles to Latex and HTML.

In spite of having omitted or glossed over many of the specifics, this overview should have been sufficient for communicating the flavor of the language. It is designed in the spirit of Lisp but it differs in three major respects:

- The use of variables as a special kind of symbol, thereby avoiding almost entirely the use of the quote operation. This brings the language closer to the conventional notation of mathematics and logic, and evaluation becomes a special case of partial evaluation.
- The separation between evaluation of expressions (including actions) from the execution of actions.
- The use of four kinds of brackets, leading to considerably improved legibility of formal expressions as compared with S-expressions that use round parentheses only.

Entityfile Syntax An *entityfile* is a sequence of entity-descriptions and is usually stored as a textual file in the sense of the file system being used. Each entity-description contains information about an entity (i.e. a symbol or a composite entity), in two ways:

- A set of *attribute assignments* each consisting of an attribute and a corresponding, structured value which shall be a KR-expression.
- A set of *property assignments* each of which consists of a property and a corresponding, textual value which shall be a string, usually having several lines.

The syntax for entityfiles is designed in such a way that it can conveniently be used for a variety of information, ranging from object-oriented databases to files containing procedure definitions in a programming language. For example, a counterpart of the PDDL language is easily obtained by organizing entityfiles and the entities in them in a suitable fashion. Also, of course, absolutely all definitions for the Leonardo system are expressed as entityfiles.

By way of comparison, the use of composite entities and of arbitrarily complex KR-expressions as attribute values helps to avoid much of the clumsy reification of composite expressions that is a consequence of using RDF-style triples. From the point of view of reading convenience, entityfile syntax is much superior to just stating information as a long list of propositions, as is done e.g. in SUMO files.

Open Educational Resource

The CEL representation language as well as other aspects of the Leonardo design are used in a modular textbook for A.I. called *Artificial Intelligence – A Coherent Approach* that is being prepared as an Open Educational Resource. Although some chapters in it are still not ready, it is anyway sufficiently complete that it has been used for several A.I. courses at Linköping university. These learning materials are available on a website, [4]

I propose that it is very valuable to have a notation for knowledge representation that can be used both in textbooks and other publications, and by the software platform for A.I. systems, since this facilitates bringing published methods to their implementation, and it also facilitates the display and publication of examples of the system's behavior and performance.

A Roadmap-Based Paradigm for A.I. Research

In addition to the description of the Leonardo system, the Leonardo Project, and the underlying design decisions, I also wish to comment on whether the current A.I. research paradigm is adequate for the goal that is addressed by the present workshop.

The Return to the Original Goal

This workshop is dedicated to the original goal of Artificial Intelligence research, namely, the construction of a software system that *has intelligence* in a sense that is commensurate with how humans have that property. The organization of the workshop may be interpreted in two ways: as an indication of rapid progress towards that goal in current research, or as a sign of dissatisfaction and frustration that progress is so

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

slow in this particular aspect of A.I. research. I subscribe to the latter view and the following observations and proposals are made on that assumption.

Artificial Intelligence is often presented as a collection of theories and methods that are studied independently of each other, together with a vague long-range vision of building artificial systems that have broad-based intelligence. Current projects in the field are supposed to contribute towards that long-range goal by developing and testing methods, as well as theories, that can later on be combined into an overall system. When some contemporary A.I. projects aggregate several current techniques in the systems being developed, this could hopefully be early stages in an aggregation process that will ultimately result in achieving the long-range A.I. goal.

The following are some often discussed problems with our present style of research:

- Too much theory, not enough work on systems
- Contributions do not build on each other to a sufficient extent
- Difficult to publish systems-related work. Current publication formats discourage reports of the important aspects of experimental system design, and favor reports of aspects of marginal importance.

I have the following proposal for how these problems can be addressed, and for how we may be able to move more rapidly and more efficiently towards the workshop goal.

First, the field should formulate a *vision* of what the target system is to be like, or a few competing visions. McCarthy's original "advise taker" proposed such a vision, but it has not been strong enough as a guiding principle. The SOAR architecture and the BDI model are other examples of such visions. The concept of an "intelligent software individual" as described in this article is one more proposal. It differs from the preceding ones in particular through its emphasis on representation languages and on integration with basic software engineering techniques (programming languages, operating systems, database systems).

Secondly, each such vision should be associated with a *roadmap* for how to achieve the vision, so that research contributions can be presented in terms of how they contribute to one or more of the recognized roadmaps. The sequence of abstract design decisions for Leonardo may serve as an example of such a roadmap. If done in the right way the use of roadmaps will facilitate for readers (and for reviewers) to assess the relevance of proposed contributions. It may be that at present, research contributions are evaluated too much in terms of their perceived "depth" and "originality," and not enough in terms of how they contribute to a goal. This is understandable if the goal and the roadmap to the goal have not been identified in a clear way, but it is a problem that can be corrected.

Of course, it is neither possible nor even desirable to identify a single vision and a single roadmap that everyone subscribes to, and it will be perfectly fine to have several of them. However, it will help a lot if visions are made explicit and if they are debated and compared, and if each publication is required to relate to at least one of the field's major roadmaps – or to make a solid proposal for a new one.

Joint Development of Software Resources

I have described the Leonardo system in terms of a sequence of abstract design decisions, and I have also proposed that a similar set of decisions could be used as a roadmap by a community of researchers. However, there are some important differences between the use of such a set of design choices for a specific project or as a roadmap.

For example, regardless of context, the development of a software platform should be followed by the development of a library of software resources that are based on that platform. In the context of a particular project it will be natural to identify a specific set of resources that are to be implemented, and in the proposal for a research project one will often wish to include some innovative resources of new kinds. From the point of view of a community effort, on the other hand, the first step towards such a library should be to implement techniques that are well established in the A.I. research literature in such a way that they work well together with the software platform, and to use them in an explorative fashion for a number of moderately sized projects. The experience from those projects should guide the continued work on the library.

The following are a few obvious examples of desirable contributions to such a library, but there are many other candidates for this list as well.

- Support for learning and use of decision trees.
- Support for the learning and the use of causal nets/ Bayesian nets.
- Prediction of short-range and longer-range future states of the individual and its environment, based on causal information expressed in logic and/or in causal nets.
- Facilities for action planning and for the robust execution of actions.

It may be possible to view the overall development effort as a case of open software development, and to organize it in the same way as other open software efforts. However, from an academic research point of view it will also be important to have a publication mechanism for reviews, evaluations and systematic comparisons of software resources as well as for the concise descriptions of those resources themselves. Existing publication venues in the A.I. field may not be willing or able to provide that service.

References.

Apologies for the lack of explicit references, due to the space restrictions. The full article should contain correct references to languages and systems that are mentioned in the text, including Lisp, KIF, KQML, PDDL, CYC, SUMO, BDI, SOAR, and HTN. On the other hand, since these are very well-known and sources for them are easy to find, for example using Google or Wikipedia, the reference list will mostly be a proforma thing.