Reification of Action Instances in the Leonardo Calculus

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

Abstract

This article describes the Leonardo Calculus, a logicist representation for actions and change where action instances are reified. This makes it possible to represent relationships between actions, such as the relation between an action and its sub-actions in a hierarchical action structure. It also makes it possible to represent the relation between an action and the objects that have been created in it, for example, the relation between the action of building a house and the house itself, or the relation between the action of understanding a sentence and the parse tree that was an intermediate representation for the sentence.

Since action instances are reified, the logic must limit the set of action instances in each model to those that are actually warranted by known knowledge, which suggests minimizing the domain of action instances, rather than one or more predicates. However, somewhat surprisingly, this calculus ends up minimizing a predicate anyway.

1 Introduction

Logics of actions and change are interesting for a number of practical purposes, including both natural-language understanding and cognitive robotics. In any such application area there needs to be an interdependence between the overall design of the computational system at hand, and the 'design' of the logic, including both its syntax and its semantics. Usually the interdependence has been one where the logic was designed first, and its use in an implemented system came afterwards.

In the work reported here I wanted to proceed in the opposite direction, starting with the design of a clean and simple executive for cognitive robots, and then asking what are the ramifications of that design for the choice of logic of actions and change. However, when looking at the outcome of the implementation experiment, I had a certain sense of *déjà vu*, since the structures that had naturally evolved while implementing the system seemed to provide an answer to some issues that I had previously been thinking about from a theoretical perspective.

The approach to logic for actions and change that is to be described here can therefore be presented both from the point of view of logic design, and from the point of view of the implemented system that provided the concrete inspiration.

2 The Logicist Perspective

Several current approaches to reasoning about actions and change choose to represent time explicitly as argument(s) of predicates such as *Holds* and *Do*, which are used for expressing that a certain property holds in the world at a certain point in time, or that a certain action or event occurs during a particular interval between two timepoints. This approach was first introduced by Shoham [8], and has been pursued e.g. by Shanahan et al with the 'modern event calculus' [7], by Sandewall in the 'Features and Fluents' approach [5], and by Doherty et al with 'Time and Action Logic' (TAL) [1].

Logics in this category usually define one or more ways of minimizing the changes of fluent values that occur during a chronicle, that is, a history of the world as described by the logic. In principle they should also minimize occurrences of actions or events themselves, for example by minimizing the extent of the *Do* predicate, since axioms characterizing the effects of actions will usually be implications from the occurrence of an action, to the occurrence of one or more changes of value for state variables. If action occurrences are not minimized then one will obtain unintended models involving unmotivated action occurrences, together with the value changes that they imply.

Minimizing both value changes and action occurrences is tedious, however, and there is a well-known way of avoiding it, namely, by considering the *Do* predicate, as used for a particular action, as an abbreviation for the action effect law for that same action. In this way the use of the *Do* predicate is taken outside the logic proper, and one obtains the effect of only accepting those action occurrences that are explicitly stated in atomic propositions ("ground unit clauses") using that predicate.

However, this solution works only if the *Do* predicate is only used in fairly simple propositions; it fails to work if, for example, there are axioms of the form

```
Do(s,t,a)  imp  exists u [Do(t,u,e)]
```

which are obtained if causation is represented as a connection from one action/event to another action/event, and not merely as a connection between two value changes for state variables. In such cases it seems necessary to use the more general solution of treating *Do* as an ordinary predicate, and to minimize its extension according to some policy.

Now comes the essential point for the present article: If we are anyway going to minimize on action occurrences, which is what we do by minimizing *Do*, then we may as well reify action occurrences and minimize that object domain. The advantage with doing that is that it makes it possible to state other information about action occurrences, such as subaction-superaction relations, result relations, and so forth. In particular, we have noticed in earlier projects that it is sometimes important to be able to specify results that are obtained from an action, in particular for cognitive actions that occur within an agent. For example, the cognitive action of parsing a phrase in natural language may be considered to have a parse tree as a result.

This general argument suggests that it is the domain of action instances that ought to be minimized in such an approach. It turns, out, however, that in a technical sense one must anyway minimize a particular predicate, although the intuition is still the one of minimizing a domain.

This is the essence of our contribution from the formal-logic point of view. Let us now switch to the system-based background for the proposal.

3 The Leonardo System

Leonardo ([1]) is an experimental software system where I attempt to integrate facilities that otherwise appear in, and are duplicated in operating system, programming language and system, intelligent agent system, dialog engine, and several other kinds of software. It has the character of a platform that can be used as a basis for various kinds of applications. One of the built-in facilities in the Leonardo platform is the support for the execution of plans consisting of several actions, and in particular for distributed execution of those plans.

The general framework is that we have *actions* and *goals*, and goals are realized using *plans*. Actions have the following characteristics:

- They may have a duration in time and execute concurrently.
- They are hierarchical; actions can have subactions.
- Actions may succeed or fail. If a subaction fails then the higher level action that it serves has to proceed differently in order to remedy the situation.
- Actions are performed by agents, and an agent may delegate an action to another agent for execution. The delegee reports back the outcome when the actions terminates.

All of this is of course standard fare from the point of view of intelligent agent systems. Notice, however, that it is not at all clear how to represent all of these phenomena in a logic of actions and change, and that is the problem that is being addressed here.

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

3.1 Representation of actions in Leonardo

Leonardo is based on a textual representation of data that plays the same role there as S-expressions do in Lisp and that is called LDX, for Leonardo Data eXpressions. LDX can also be compared with semantic web notations such as RDF and OWL. It is used for almost everything within the system, and in particular it is the basis for the Leonardo Calculus for actions and change. We shall introduce it through an example, namely, the information that is used and produced when the system executes a plan consisting of three actions. This example has been obtained from an actual run with the present Leonardo system.

The following is a method description in Leonardo:

```
-------------------------------------------
-- method4

[: type method]
[: plan
 {[intend: t1 t2 (remex: lar-001-004
                   (query: whatyourbid))]
  [intend: t1 t3 (query: whatbid)]
  [intend: t4 t5 (query: whatprop)]}]
[: time-constraints
                  {[afterall: {t2 t3} t4]}]
-------------------------------------------
```

This represents an entity called `method4` that has a number of *attributes*, in particular the attribute for `plan`. Its value is a plan, i.e. a kind of high-level procedure, for performing the action `query:` three times with three different arguments. The time when the first two occurrences are to start is called `t1`; the third occurrence starts at a time `t4` which is defined as being when the first two occurrences have ended. The time when the first mentioned occurrence ends is called `t2`, and similarly for `t3`. The method consists of a set of intended actions, and set of time constraints between them.

This plan is supposed to be executed in a particular agent (called `lar-001-003` in our specific run of the plan) but the first mentioned action is to be remote executed (therefore `remex:`) in another agent called `lar-001-004`.

The representation that is used for expressing the plan is general-purpose in the sense that it is used for both procedures and data, as well as for information with an intermediate status such as the 'plan' above. It is organized in terms of *entities* each of which has a number of *attributes*. In our example, there is an entity called `method4` with three attributes `type`, `plan`, and `time-constraints`. All entities must have a `type` attribute, and the value of this attribute determines what other attributes may be present.

Attribute values are expressions in the Leonardo data expression language (LDX), which is a textual representation of the data structures that are used within the system. Expressions in LDX may be atomic ones (symbols, strings, or numbers), or may be formed recursively using the operators <...> for sequences, {...} for sets, `[...]` for records, and `(...)` for forming composite entities. In the example, `(query: whatbid)` is a composite entity that has a type and attributes, just like the atomic entity `method4`. These types of expression can be nested to any depth.

The use of triples of the form

```
(entity, attribute, value)
```

is part of a long tradition in AI and elsewhere in computer science. In comparison with e.g. OWL ([2]), LDX is distinguished by allowing composite entities as the carriers of attribute-value pairs, and by allowing several kinds of structures (sets, records, etc) in the attribute values.

The LDX language is used for representing both procedures and data, both within the system and for the applications. In particular, it is used for the ontology that is a backbone in the organization of an agent([3]) in Leonardo. It is also used for representing the 'systems' data structures that are needed for administrating the execution of computational processes.

3.2 Attributes of actions

The Annex displays an interactive session where this plan is put to use, and in what follows we shall refer to the Annex for examples.

Actions are hierarchical, so actions can have subactions, or more precisly, each action instance can have sub-action-instances. In our example, a `query:` action invokes an `ask:` subaction that makes the prompt and receives the answer. If the answer is malformed then `query:` asks the user again until a correctly formed answer is obtained.

Each action instance has a starting time and an ending time, and is *operating* between those times. When an action instance or goal instance terminates, it obtains an `outcome` attribute and an `endtime` attribute. The latter is the timepoint of termination. The `outcome` attribute represents whether the action *succeeded* or *failed*, using records beginning with `result:` and `fail:`, respectively. Result records can report a 'value' that results from the action, as well as ancillary information; fail records can report the character of, and possibly the reasons for the failure. The outcome of an action is reported to the superaction from which the current action was invoked, or the goal instance invoking it, or the other agent invoking it in the case of remote execution, or a combination of these.

Pursuit of a goal is straightforward if there is an appropriate plan and if all the actions in the plan succeed. If some action fails, and unless a remedy for the failure has already been defined in the plan, then replanning or resort to the user must follow. Replanning has not been implemented in the current system.

3.3 Robotic actions

The example in the Annex is based on actions that prompt the user for input and return that input as the result. Robotic actions, on the other hand, are actions that the executive must 'visit' at regular intervals while they are operating, in particular for receiving sensor data and for producing control information to actuation subsystems. The Leonardo platform does in fact contain an action executive that does this as a generic service, besides listening to commands from user input and those coming from other individuals. It is therefore straightforward to define actions that map incoming sensor data to outgoing actuator data in each cycle during their execution period.

Our example here does not illustrate the support for such robotic actions. However, the example has still been sufficient for exemplifying all the aspects that were mentioned in the introduction, in particular since success/failure of actions is simulated and concurrency is represented.

4 The Leonardo Calculus

The LDX notation is the textual representation for many kinds of information in the Leonardo system, ranging from low-level system data to ontologies and other knowledge structures. We have seen already how LDX is used to express, for example, the information about the superaction-subaction relationship, as well as their relations to the stated goals of the individual, their success or failure, etc.

The *Leonardo Calculus* builds on LDX and provides a logicist framework for representation and reasoning about actions that is closely connected to how actions are treated in the Leonardo system. This has a certain novelty value since no presently used logic for actions and change is capable of representing this kind of information about actions.

Notice, by the way, that the control information in the example that was shown above and in the Annex, is represented using structured data that are available for further processing, whereas in a conventional programming language or other conventional settings the same kind of information would be intrinsic to the run-time system or executive, and hidden from the user-programmer. This is exactly what distinguishes systems of this kind (intelligent or knowledge-based agent systems) from conventional programming languages. Furthermore, the control information is not only available, but there is also this textual representation for it in terms of the LDX syntax. This has laid the groundwork for making the control information available to a logic and a reasoning system.

The reason why presently used logics do not suffice for this task, is that the control information involves relations between individual action instances, as well as between action instances and other data, such as the goals they serve, or their results. This is not expressible if actions are essentially propositions, as in explicit-time logics (TAL, CRL, modern event calculus, etc). It is also not expressible in the situation calculus, and there one has the additional problem of expressing concurrent actions with overlapping durations in a clean way.

4.1 The Formal Language

We use first-order predicate calculus as the host language, and configure it to our needs through the introduction of suitable predicates and functions which will be defined below. For the concrete syntax, terms are represented as LDX entities, and atomic propositions are represented as LDX records. This means that terms are written using round parentheses, and atomic propositions are written using square brackets; in both cases the function symbol or predicate symbol is written prefixed and inside the brackets, like in Lisp.

[2] `http://www.w3.org/TR/owl-features/`

[3] The term used in Leonardo is an *individual*; individuals are somewhat similar to 'intelligent agents'. The words agent and individual are used as synonyms here.

Logical connectives are written as infixes in the obvious ways, using the symbols `and`, `or`, `imp`, etc. We use standard keyboard characters in order to stay close to the implementation. An expression of the form `A and B imp C and D` is interpreted as `(A and B) imp (C and D)`, where `and`, `imp` (for implies) and the other connectives are as usual.

Atomic terms consisting of one or two characters are taken to be variables. Universal and existential quantifiers are used as usual, and free variables in formulas are Implicitly considered as Universally Quantified (IUQ). In fact, all the examples of the calculus in this article can be expressed using IUQ and without explicit use of quantifiers.

4.2 The Types and their Constructors

We assume the use of the following main types: timepoints, objects, features, values, assignments, actions, and action instances. The *timepoint* domain is constructed using the initial timepoint and a successor function, and for convenience the non-negative integers are chosen as timepoints. However, there is no assumption of metric property in it. The *object* domain is a finite set and it is assumed that each object is denoted by exactly one constant symbol. *Values* can be e.g. strings or numbers. *Assignments* can only be formed using the function `v` that maps a feature and a value to an assignment. Similarly, *action instances* can only be formed using the function `b` that map a timepoint and an action to an action instance, intended as the instance of the action that starts at the timepoint. *Features* and *actions* are obtained as the values of application specific functions of zero, one or more arguments.

Functions whose values are in the 'value' domain have their standard interpretations, for example for arithmetic functions. All other functions are considered as Herbrand functions, i.e. two terms are equal iff they are formed using the same function and their arguments are pairwise equal.

4.3 The Predicates

The main primary predicates are as follows (one more to be added below). All of these have a timepoint as their first argument:

- `H` specifies that an assignment holds at the given timepoint. This is like the traditional `Holds` predicate. A feature can have at most one value at any one time.
- `N` specifies that there is no value at the given time for a given feature.
- `F` specifies that an assignment holds from the given timepoint and onwards.
- `W` specifies that at timepoints before (i.e. strictly less than) the one given in the first argument, no value is assigned for a given feature.

Here are some examples of the use of these predicates:

```
[H 14 (v (price house4) 2200)]
[N 14 (price house4)]
[F 14 (v (nationality john) swedish)]
[W 14 (nationality john)]
```

The first example says that the price of `house4` is 2200 units at time 14. The second example says that it does not have any value at time 14. The third one says that the nationality of `john` is `swedish` at all times greater than or equal to 14, and the fourth one that the nationality of john does not have any value at any time strictly less than 14. Either it was not assigned, or John did not even exist.

The fifth and final predicate is called `E` and is used for characterizing the existence of things, in a broad sense of things including both (physical) objects, features, and action instances. The `E` can therefore be read both as *exists* and as *event*. It has an action instance as its single argument, as in the following example:

```
[E (b 16 (moveto object4 place6))]
```

which says that the *action* `(moveto object4 place6)` has a particular *action instance* that begins at time 16. It is possible of course to reason in general about the action `(moveto object4 place6)` which can have several instances if it is done several times with different starting time. It is also possible to form terms `(b` *t* `(moveto object4 place6))` for arbitrary *t*, but it is only when an instance of that action actually starts at time *t* that the `E` predicate is satisfied.

The use of functions `v` and `b` is a bit inconvenient, and the following abbreviations may therefore be useful when writing out concrete scenario descriptions:

```
[V t f x]         for    [H t (v f x)]
[S t f x]         for    [F t (v f x)]
[B t a]           for    [E (b t a)]
```

The recommended mnemonics are: H for Holds, F for From, N for Notdefined, W for wait, V for Value, S for Since, E for Exists or Effectively.

4.4 The Feature-Valued Functions

Some useful functions have an action instance as the argument and a feature as a value. In particular, the function `endtime` has as value a feature representing the ending-time of an action instance, and can be used as in this example:

```
[S 18 (endtime
         (b 16 (moveto object4 place6)))
      18]
```

which says that the action-instance that started at time 16 and where `object4` is moved to `place6`, has 18 as its ending-time, and the attribution of its ending-time applies from time 18 and onwards to infinity.

The 'do' predicate in traditional logics in the Features and Fluents or event-calculus traditions can then be considered as an abbreviation, as follows:

```
[D s t a]  for  [S t (endtime (b s a)) t]
```

5 Leonardo Calculus Expressivity

The following are some examples of how the Leonardo Calculus is used for commonly occurring representation needs when reasoning about actions. Notice that `[= x y]` is our way of writing `x = y`.

Representing Preconditions

The condition that the feature `f1` has the value `v1` is a sufficient condition for the applicability of the action `a`, i.e. if the action `a` is invoked at time `t` and the precondition holds at that time, then an action instance for `a` at `t` will start to exist:

```
[H t (invoke a)] and [V t f1 v1]
   imp [B t a]
```

Representing Termination Conditions

If the action `a` executes since time `s` and the feature `f2` has the value `v2` at time `t`, then the action terminates at that time:

```
[= ai (b s a)]  and  [W t (endtime ai)]
   and [V t f2 v2]
   imp [S t (endtime ai) t]
```

The second precondition is needed so that the action is only inferred to terminate the first time that the termination condition applies, and not at succeeding timepoints where the termination condition itself continues to apply.

Creation of objects

The following is an example with an action where `agent2` creates a new object of type `house`:

```
[= ai (b s (create agent2 house))]
  and [E ai]
  and [S t (endtime ai) t]
  imp [B t (result ai)]
  and ([V t (result ai) x]
         imp [B t (type x) house])
```

Notice that it would not be correct to use `(endtime ai)` as the first argument in the two occurrences of the `B` predicate. The expression `(endtime ai)` stands for a *feature*, i.e. something that can be assigned a value, and not for that value itself.

For the same reason, it seems to be necessary to introduce the variable `x` on the last line, for the object that is the value of the feature `(result ai)`. (However the full details of this requires going into nonobvious aspects of the semantics). This way of writing is admittedly a bit clumsy, and it may be necessary to find a more convenient notation, for example using an additional abbreviation.

Causality between actions and natural events

Assume for the moment that actions (performed by an agent controlling the physical parameters of some object) and natural events (not performed by an agent in that sense) are represented in the same way, as described above, and consider the case where the occurrence of an action `a` always causes the immediate occurence of a natural event `e` that begins at the time when the action ends. This can be expressed as follows:

```
[E a] and [S t (endtime a) t] imp [B t e]
```

6 Model-Theoretic Aspects

The Leonardo calculus is a first-order calculus from the syntactic point of view, but it has the peculiarity that it considers some things to 'exist' for limited periods of time only. This leads to a choice between two ways of structuring the semantics: either the 'exist' in this sense is identified with the existence concept in the logic, as used by the existential quantifier, or else one uses a conventional semantics where the domains are ([4]) defined irrespective of time, and where the 'existence' for limited periods is represented as a predicate. In the former case, the operator `E` that was introduced above is not a predicate in the true sense but another kind of operator; in the latter case it is an ordinary predicate.

We prefer the latter one of these choices. As a result we obtain two levels of semantics, in the same way as e.g. in 'Features and Fluents'. The *classical models* for a given set of propositions in the Leonardo Calculus are defined in the standard way using what has been said above e.g. about the Herbrand property of most of the functions. The *intended models* for a set of propositions are a subset of the classical models, and they can be constructed using a kind of simulation of a chronicle in the world being described.

For example, if the given axioms contain an action-level causal rule of the form that was shown above (rewritten):

```
[E a] and [D s t a] imp [B t e]
```

(where `a` and `e` must actually be subexpressions) then every simulation containing an action instance of the form `a` must let it be immediately followed by an event of the form `e`, in order to be a member of the intended set of models for the given axioms. The definitions of the construction process for intended models are essentially trivial.

Finally, we arrive at the question of how to characterize the set of intended models, and in particular whether this is best done in the classical way using a set of additional axioms, or whether a nonmonotonic approach is appropriate. I find the latter case more plausible and am using it as my working hypothesis unless and until a classical treatment is found. In this approach, it is clear that the predicate `E` has to be minimized according to some reasonable preference relation, in order to suppress models containing unintended actions. The feature-oriented predicates `H`, `N`, `F`, and `W`, on the other hand, do not offer any particular new difficulties. A priori it seems that the use of chronological minimization is the most plausible for the `E` predicate.

The representation of the change of value for features, either due to the effects of actions, or due to direct causation between fluent changes, can it seems be represented using occlusion in the same way as for standard explicit-time logics, as analyzed in 'Features and Fluents'.

7 Related Work

None of the current, logicist approaches to reasoning about actions and change uses reification of action instances, as far as we are aware, and in that sense the approach proposed here is new. Traditional approaches to 'semantic nets', for example for representing natural-language information have often represented action instances as 'nodes' in their conceptual graphs [3], which may be thought of as a kind of reification. The problems of formal and systematic approaches to reasoning were addressed e.g. in Sandewall [4] and by Schubert [6], but rarely arrived to the point where one could reason effectively about actions, plans, and their effects.

[4] Notice that we have several domains.

One aspect of the proposed Leonardo calculus is that the ending-time of an action is considered to be one of its attributes. The action instance is uniquely characterized by the action and the starting-time; this differs from standard explicit-time approaches where the proposition representing an action instance involves the starting-time, the ending-time, and the action. The new way of looking at this is actually well in line with the theoretical analysis of nonmonotonic entailment rules that was done in 'Features and Fluents', since the proper treatment of the ending-time was complicated in the treatment in several of the entailment rules there. It was intuitively clear, already there, that the ending-time of an action instance should be viewed as an attribute of the action instance, and not as an intrinsic part of it.

Hayes' early work on frames [2] addressed the use of sets of attribute-value pairs for the representation of knowledge, and defined the directions for the use of concept languages and for the contemporary development of representation languages for the sematnic web, in particular through OWL. The LDX representation language that is used in Leonardo belongs to the same tradition, but adds the use of several additional kinds of structures.

8 Software Architecture Aspects of the Leonardo System

This article has emphasized a proposal for a non-standard approach to reasoning about actions and change, which is motivated both from a principled point of view and as the outcome of a particular explorative programming project. Another aspect of the same work, although not discussed in the present article, is that it provides an approach to the design of realistic software systems that are closely tied to the use of a particular logic or calculus. The Leonardo system as such is an experiment with a new way of organizing the basic software in the computer – operating system, programming language system, etc – in a better structured way than before. There are many aspects to that work, but one of them is that this new architecture integrates some of the characteristic features of intelligent autonomous agents into the system core. As this article has shown, not only the software practices of intelligent agents, but also a formal calculus for actions and change has been placed in that central position of the software system.

9 Annex

The following is a commented log of an interactive session with Leonardo. It is the log of an actual run of the system which has been post-edited so as to omit some parts of the entity descriptions in order to avoid irrelevant details. Additional line breaks have been inserted in order to fit into the two-column format.

There are two open command-line windows on the computer screen or screens, one for each of the two Leonardo individuals `lar-001-003` and `lar-001-004` which may be located on the same computer or on two different ones.

The interactions on `lar-001-003` go as follows, after the obvious startup of the system:

```
066-> adg (achieve: demo example A)

067-> selmeth method4
```

Each interaction is numbered; user input consists of a command often followed by an argument. The `adg` command requests the system to adopt a particular goal which is characterized by the command's argument. In the full system this should lead to a process for obtaining a plan, either by planning from first principles or by retrieving a plan from an archive. In our demo we have shortcut this by the second command, `selmeth`, which simply instructs the most recently introduced goal which plan to use. The plan starts to execute when the user enters the command `seg`, for 'start execute goal':

```
068-> seg
    > (adogoal: 66 (achieve: a b c))

069=>
----> What is your bid? 16200

070-> Continuing:
AI (b: 68 (query: whatbid)) completes:
Succeed, result: 16200

071-> Continuing:
Received outcome f action started at: 68

072=>
----> What is your proposal? 13000
073-> Continuing:
AI (b: 71 (query: whatprop)) ready:
Succeed, result: 13000
Completed goal:
     (achieve: a b c) adopted at: 66
```

The following is what happens. When the user types in `seg`, the action `(query: whatbid)` starts to execute in the individual at hand, which has the effect of displaying the prompt `What is your bid` in the individual's user dialog. At the same time, the first action in the plan starts to execute remotely, in the other individual, where it displays the prompt `What else do you want to say?` on its screen. The wordings of the prompts is obtained because the arguments of `query:` are separately defined entities that have the wording as an attribute. The following is the definition of the entity `whatbid`:

```
---------------------------------------
-- whatbid

[: type output-phrase]
[: englishphrase "What is your bid?"]
[: swedishphrase "Vad r Ditt bud?"]
---------------------------------------
```

The user for the first individual answers the prompt with the value `16200`, which counts as interaction `069`, and the system confirms completion of that action in interaction `070`. The first individual also receives the value from completion of the action in the other individual, in interaction `071`. The top level executive listens to input both from the user and in channels from other agents/individuals.

The completion of the first two actions allows `lar-001-003` to start performing the third action in the plan, leading to interaction `072`, after which the goal is reported as completed in interaction `073`.

The information about what actions were performed, for what reason, and with what results, is represented as LDX data structures and is therefore available for inspection and for further processing. The command `log`, for 'list old goals', displays the current information about the goal used above, as follows:

```
076-> log
(adogoal: 66 (achieve: a b c))
  Plan name:  method4
  Plan:  {[intend: t1 t2
              (remex: internal-ch-02
                        (query: whatelse))
                     :done t]
           [intend: t1 t3
              (query: whatbid) :done t]
           [intend: t4 t5
              (query: whatprop) :done t]}
```

This is like above, except that each of the actions has been marked as completed. The format of the logs is a slightly sugared variant of LDX. The command `loa`, for 'list old actions' displays the actions that were performed in the first individual, as follows:

```
075-> loa
  (b: 68 (query: whatbid))
      Towards goal
         (adogoal: 66 (achieve: a b c))
      State          [result: 16200]
      Subactions
         <(b: 69 (ask: whatbid))>
      Outcome        [result: 16200]
      Endtime        71
  (b: 68 (remex: lar-001-004
               (query: whatelse)))
      Towards goal
         (adogoal: 66 (achieve: a b c))
      State          [requested:]
      Subactions     <>
      Outcome        [result: 12900]
      Endtime        72
  (b: 69 (ask: whatbid))
      Subaction-of
         (b: 68 (query: whatbid))
      Outcome        [result: 16200]
      Endtime        70
  (b: 71 (query: whatprop))
      Towards goal
         (adogoal: 66 (achieve: a b c))
      State          [result: 13000]
      Subactions
         <(b: 72 (ask: whatprop))>
      Outcome        [result: 13000]
      Endtime        74
  (b: 72 (ask: whatprop))
      Subaction-of
         (b: 71 (query: whatprop))
      Outcome        [result: 13000]
      Endtime        73
```

The functions `adogoal:` and `b:` are further examples of functions that form composite entities. The function `b:` takes two arguments, namely a timepoint and an action, and forms an entity for the action instance that is/was invoked at the time given in the first argument. The function `adogoal:` is similar but it forms a goal instance from a timepoint and a goal, representing the particular goal instance that results when the goal is adopted at a particular timepoint.

References

[1] Patrick Doherty. Reasoning about actions and change using occlusion. In *European Conference on Artificial Intelligence*, pages 401–405, 1994.

[2] Patrick Hayes. The logic of frames. In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 46–61. De Gruyter, 1979.

[3] Ross Quillian. Semantic memory. In Marvin Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, 1968.

[4] Erik Sandewall. A set-oriented property-structure representation for binary relations, SPB. In *Machine Intelligence 5*. Edinburgh University Press, 1970.

[5] Erik Sandewall. *Features and Fluents. The Representation of Knowledge about Dynamical Systems. Volume I.* Oxford University Press, 1994.

[6] Lenhart K. Schubert. Extending the expressive power of semantic networks. In *Proceedings of the Fourth IJCAI*, pages 158–164, 1975.

[7] Murray Shanahan. Prediction is deduction but explanation is abduction. In *International Joint Conference on Artificial Intelligence*, pages 1055–1060, 1989.

[8] Yoav Shoham. *Reasoning about Change: Time and Causation from the Standpoint of Artificial Intelligence*. PhD thesis, Yale University, 1986.