# Actions as a Basic Software Concept in the `Leonardo` Computation System

**Erik Sandewall**
Department of Computer and Information Science
Linköping University
Linköping, Sweden
erisa@ida.liu.se

## Abstract

The work reported here is performed in a broader context where we propose to change the overall software architecture (operating systems, programming languages, etc etc) in order to eliminate the considerable redundancy of concepts and constructs that contemporary software technology exhibits. This requires, among other things, a realignment so that some constructs that used to be placed on higher levels of software now become incorporated in a kernel on a much lower level.

In this framework, we propose in particular to use the construct of an *action* already in the kernel, whereby it becomes available for many applications as a conceptual and computational resource and in a uniform fashion. The article describes and discusses the ramifications of this approach, including how it relates to the current state of the art in logics of actions and change, as well as the non-monotonic character of one of its computational constructs.

The article has been written for the purpose of a workshop, so its contribution is in the range of ideas and as a discussion-starter. It does not pretend to report finished results.

## 1 Actions as a Basic Software Concept

Actions, in the sense of a process that changes its environment in a describable, and often goal-directed way, occur in several kinds of software systems. They are important constructs in 'intelligent agents' and they also occur in some simulation systems, for example. However, in all of these cases the actions occur in a relatively high layer of the overall software architecture of the computer. Lower layers include the operating system, the programming language and its environment, communication systems such as CORBA or OAA, possibly combined with database systems, and so on.

In this article we propose to change things around so that actions are introduced in a much lower level of the overall software architecture, in a way whereby they become available for many applications as a conceptual and computational resource and in a uniform fashion. This proposal is part of a broader idea concerning software reform in order to integrate the traditional concepts of operating system, programming language, database system, document formatting system, and several others. The reason for doing this is that the traditional overall structure of software contains a lot of conceptual redundancy: similar, but not equal conventions and constructions are introduced in different parts of the overall system. It ought to be possible to design the system in such a way that this redundance is eliminated.

We are in the process of designing an experimental language and system, called `Leonardo`, for the purpose of investigating the feasibility of such a reform. One important aspect of `Leonardo` is that actions occur already in the system kernel. In fact, the proposed reorganization tends to change things around in more than one way, so that things that used to be thought of as 'high level' now become incorporated in the kernel, which is a natural consequence of the desire to remove conceptual redundancy.

The present article will first give a quick overview of the present version of `Leonardo`, with particular consideration of its action facilities, and then proceed to a discussion of how this relates and may relate to research about actions and change. Since this article is intended for a workshop, it combines presentation of some results achieved with a discussion of design issues that are still somewhat open, and of possibilities for future development in the cross-section between programming systems on one hand and logics of actions and change on the other.

The name of 'Leonardo' was chosen after Leonardo da Vinci, and since we believe in the need for a renaissance in software technology - a renaissance where many existing dogmas are rejected and where we return to some of the concepts that were invented long ago but have been forgotten meanwhile.

## 2 Functional Aspects of `Leonardo`

The `Leonardo` representation language can be thought of both as a programming language and as a knowledge representation language, and in fact it should also be used for the purpose that is commonly served by the 'shell' command language for the operating system. We believe that a single language kernel should be used for these purposes and several others, albeit with variations that adapt it to interpretive or compiling environments. The basic design considerations are:

- to stay as close as possible to the notation of set theory and other discrete mathematics, while using the 8-bit standard ASCII character set,
- to favor powerful, orthogonal concepts and constructs,
- to view the entire Internet as the logical 'memory' (resource for storage and retrieval of data) of the language.

This language has a functional aspect and an action/agent aspect. The present section contains an outline of the functional aspect.

## 2.1 Expressions

The basic data thing in `Leonardo` is called an *expression*. We say 'data thing' since objects are another kind of thing. There are *data expressions* and *text expressions* which have different syntax, but each of them can be embedded inside the other, recursively, in some positions requiring an escape character. The following data expression:

```
[automobile: :brand Volvo :year 2005
             :type sedan]
```

is a record, presumably denoting a description of a particular car, for example as a database query. The following text expression:

```
<[style: :bold t :font ariel]
         boldface text in ariel font>
```

presumably denotes five words that are typeset in boldface ariel font. The first subexpression of the text expression is a record (hence, a data expression) specifying the formatting. Argument lists that appear in records, forms, and a few other constructs may have a few initial, untagged elements in prescribed order, followed by optional, tagged elements in arbitrary order (like in CommonLisp).

`Leonardo` allows several other kinds of data expressions besides those that were exemplified above. There are expressions for sets, sequences, mappings, texts, and a few more. We try to keep the notation as close to traditional set-theory notation as possible. The syntax for text expressions allows for basic markup. Sets can be specified both by enumeration of their members, by a characterizing property ("the set of all x such that ..."), and by standard operations on sets. Functions are viewed as mappings which are a kind of sets; recursive functions are characterized as set-valued solutions of equations in the obvious way.

We refer to sets, sequences, mappings, texts, etc as different *sorts* that are represented by data expressions, and to the variety of records that are denoted by the initial symbol in the record expression, as being different *types*. The `Leonardo` representation language specifies the syntax for each of the sorts, but types can be checked dynamically. Structure specifications for types are optional, and are not part of the system kernel.

## 2.2 Unification of Ontology and Computation

`Leonardo` expressions are intended to be used both as a representation language (corresponding to frames, XML, or semantic-web languages such as OWL) and as a programming language for defining computational processes. It is therefore adapted to ontology-oriented (i.e. data-model-based) programming, where one first defines the ontology for the application at hand and represents it formally, and then the elements of the ontology are used as carriers for the definitions of data types, procedures, and other expressions that are needed for defining and performing the computation.

From the perspective of programming languages, the `Leonardo` representation language is oriented towards functional programming, and it is set-theory-oriented in the same sense as e.g. Prolog is logic-oriented. From the perspective of representation languages, we believe that the possibility to define functions in the usual mathematical sense (mapping from arguments to values) is an important feature in any representation language, which adds to the good reasons for integrating the notations used for knowledge representation and for programming.

Besides the ontology aspect and the procedural aspect, there is also a document processing aspect in the `Leonardo` representation language. We consider documentation to be a fundamental part of any software system, and therefore documentation should be integrated as well as possible with the other aspects of the software. This is why 'text', including structured and marked-up text is also defined as expressions in the `Leonardo` representation language.

## 2.3 Locations and References

Expressions can be manipulated directly by computational processes in a `Leonardo` system, but they can also be *deposited* in *locations*. Locations are important for actions, which are our next topic, since computational actions often operate on the contents of locations. For example, the action of running a document through latex is considered as operating on a location containing both the source (.tex) and target (.pdf) version of the document, as well as other, related files.

From the programming-language perspective, on the other hand, locations are used both like 'variables' in conventional programming languages, and like filenames in a conventional programming system, and like URL:s. In fact, the entire Internet 'address space' (thinking of a URL as an address in the universal computer) constitutes most of the *location space* from `Leonardo`'s point of view.

Each location is used for a particular sort, but record locations do not make any assumption about the type of their contents, in line with the interpretive character of the kernel `Leonardo` system as a whole.

A *reference* is a formula that specifies a location and that is used, in the language, for denoting the expression that is currently deposited in that location. For example, the following reference

```
[?filerec: "C:/leo/doc/" section2 vfr]
```

denotes a record that is stored in the location `C:/leo/doc/section2.vfr` according to the deposition method ("format") specified by the name `filerec`. (This is not an actually existing deposition method, it is given by way of example). Similarly, the following reference:

```
[?textfile "C:/leo/doc/" section2 txt]
```

denotes a text in `txt` format that is currently deposited in the location `C:/leo/doc/section2.txt`; the operator `textfile` specifies how it is accessed. Record references and text references are distinguished by the fact that the deposition method of the former ends with a colon character.

In these examples the path to the file in question was represented as a string. Other methods than `filerec` and `textfile` would be used if the path is to be represented as a sequence instead.

These examples suggest that files in contemporary operating systems, such as Linux or Windows can be used as locations. This is certainly possible but not the only possibility; other basic storage facilities such as object-oriented ones may be more appropriate in the future. `Leonardo`'s representation for references allows for several possibilities.

Reference expressions can be nested, like other expressions, so one can for example write

```
[?textfile
    [?textfile "C:/leo/" curdir txt]
    section2 txt]
```

whereby the system, when using this expression, will use the contents of the file `C:/leo/curdir.txt` as a directory name which is combined with the filename `section2.txt` to obtain a file whose contents are in turn obtained as a string.

References may be passed as arguments to functions or other computational entities. In this sense they are analogous to pointers in conventional programming languages. At some point the reference has to be *resolved*, for example by obtaining the contents of a textual location as an actual text. This is often done as a single operation like if, in a conventional programming language, a text file is read into working memory at one stroke and becomes a string.

## 3 Action and Change Aspects of `Leonardo`

### 3.1 Action Expressions, Agents, and Actions

We turn now to the topic of action and change. There are three related concepts in `Leonardo`, namely *action expressions*, *agents*, and *actions*. The following is an example of an action expression:

```
[fly-to! :agent witas-4
   :destination [geo-coord: 425 862]]
```

saying that the agent `witas-4`, which is an unmanned helicopter, shall fly to the point located at geographical coordinates (425, 862). The following action expression:

```
[add-to-account!
   :agent [?account-agent@ mybank]
   :account 634422
   :amount [money: ECU 4900]]
```

says that the agent `[?account-agent@ mybank]`[1] should find some way of adding 4900 ECU to the current balance of account number 634422, by whatever means it can find of raising the money.

[1]This is a reference formula as introduced in subsection 2.3 where the @ character specifies an agent.

Each action expression must have an explicit or defaulted `:agent` field specifying what agent is responsible for performing the action. There is a variety of ways of specifying the agent, as these two examples have indicated. If an action expression is presented to its agent and that agent accepts to execute it, then *an action* arises, that is, a computational process that is usually a lightweight one and that proceeds through a sequence of *steps*. The action has a local state during and after its execution. The local state is always a record, and one part of the definition of an action specifies the next-state function that is applied in each step of its execution.

### 3.2 The Top Level of a `Leonardo` System

The top level of conventional interpretive languages, such as Lisp, is a read-eval-print loop. The top level of `Leonardo` system is instead an executive for actions, somewhat similar to an object-oriented simulator. It maintains a set of *pending action expressions* and a set of *working actions*. Pending action expressions refer to actions that have been requested to the system but that have not yet started to execute, for example because not all their preconditions are satisfied, or because of concurrency constraints. Working actions are actions that have started to execute but which have not yet arrived to a quiescent state.

In its normal main cycle, `Leonardo` first allows the user to enter an action expression and adds it to the pending set. After that, it checks for each member of the pending set whether its preconditions are satisfied, including concurrency restrictions, and if so a new action is initiated as specified by that action expression and added to the working action set. Finally, the executive visits the working actions and performs the update in each of them, according to the specifications that are given by the agent of the action in question. Actions that have reached a quiescent state are moved from the working set to an archive of past actions.

This general formulation of the top-level loop can be adapted, specialized, or extended in various ways. A conventional read-eval-loop can be implemented using an 'eval' verb whose actions always finish in one step. Backgrund tasks such as fetching information from remote websites can be set up as actions with extended duration. Simulators of, and supervisors of physical robotic equipment, as well as servers can also easily be represented in the same structure. Actions are of course allowed to invoke sub-actions, the process of the main action being conditional on the process of its subactions. The subactions need not use the same agent as the invoking action; this is our counterpart of message-passing between agents.

### 3.3 Specifications of Actions

Actions in `Leonardo` are characterized using Cognitive Robotics Logic (CRL)[5] which is a reified temporal logic[9], based on 'Features and Fluents'[4] and closely related to Doherty's Time and Action Logic[1] (TAL), and having many points in common with modern event calculus as presented by Shanahan[8].

The behavior and the effect of actions is defined by a combination of the action-verb and the agent performing the action. The *external* behaviors of actions are normally expressed in CRL and are specified in terms of preconditions, postconditions, and conditions characterizing concurrency restrictions and other aspects of intermediate states.

The *internal* behaviors of actions specify the details of their execution in terms of updates of its current state at successive (but not necessarily contiguous) timepoints. Each action has a current state, starting with the timepoint when it was initiated; this current state must always be a record. In addition, each agent has its current state, which is also a record. The next-state transformation defining one step in the execution of an action has access to both of these records and is able to update them both, modulo constraints that can arise by interference between concurrent actions. It can be expressed in a variety of ways:

- As a computational procedure
- As a finite-state or hybrid automaton
- By a combination of discrete state transitions and partial evaluations of state-expressions (details below)
- As a sequence or other temporal structure of subactions

In all these cases the behavior may refer either to computations that are performed within the computer at hand, or to actions performed by a robot under the active control of that computer, or to actions that are performed independently of the computer but are observed by it.

Definition using partial evaluation offers a structured but expressive way of characterizing 'hybrid' actions that combine gradual change of state with occasional qualitative changes. The current state of the action is expressed as a record, like for all actions in `Leonardo`. This record is an expression that may contain unevaluated forms. In each cycle, the system traverses that record, replacing variables by their values, when available, and evaluting forms (functions with arguments) when possible. Individual symbols are left as they are, and records, sequences, and sets are merely traversed, i.e. their components are evaluated but the structure of the record (etc) is retained. In particular, an expression not containing any variables or forms always evaluates to itself. After that, a set of qualitative state transitions is compared to the state at hand, and any applicable transition is performed.

## 3.4 Prediction and Planning

The design of `Leonardo` does not attempt to make it into an A.I. system from the start. Instead, the idea is to design a kernel that can be used as a platform for many common programming tasks, and one that is more powerful than what conventional software technology can offer. For this reason, the kernel `Leonardo` system does not contain full-fledged facilities for prediction and planning, but it does contain handles where such facilities can be plugged in effectively, in those applications where they are considered appropriate and useful.

We described above how the top level of the `Leonardo` system maintains a set of pending action expressions, and how in each cycle it selects those for which the preconditions are satisfied. The treatment of those action expressions for which the preconditions are *not* satisfied is a natural handle. The kernel system does not do anything about them and just leaves them in the pending set, but it is possible to define other handlers for precondition failures. Planning and plan execution capability can therefore be implemented by a routine that applies to precondition-failing action-expressions, selects a plan, and adds the plan to the set of pending actions, while considering the plan as a composite action.

## 3.5 Additional Topics

The full `Leonardo` design includes a number of additional aspects that are not covered here since they are less central for the question of the relationship to logics of action and change. Those additional aspects include, for example what to do if there is no applicable transition rule or invocation rule, and what to do if there are several concurrent action requests for the same agent. They also include questions of names, symbols, and namespace, questions of persistent objects and the use of locations, and of version management for the properties of objects. For the low-level part of robotic applications, there are questions of shared record fields or transfer of data streams, for use in the connections between sensors, controllers, and actuators on several abstraction levels. Forthcoming additional reports addressing these topics will be posted on our website (references at the end of this paper).

## 4 The `Leonardo` Timeline

Since actions are specified using Cognitive Robotics Logic (CRL) in `Leonardo` systems, each action that is performed there is characterized by two *timepoints*, namely its starting timepoint and its ending timepoint. The system also administrates *features* where, as usual, `Holds(t,f,v)` expresses that the feature `f` has the value `v` at timepoint `t`, and actions can be characterized as depending on, and affecting the values of features.

## 4.1 Timepoints during Computation Sessions

Each computational session defines a sequence of timepoints that are numbered from 0 and up, and that are related to physical time as follows. Physical time is assumed to be metric and can be measured e.g. in milliseconds. The physical timeline is divided into two kinds of intervals that alternate, namely timepoints and action-periods. The ending-time of a timepoint is the starting-time of the succeeding action-period, and vice versa, and each timepoint and each action-period is an interval on the physical timeline.

Consider in particular the case where the `Leonardo` system operates a read-invoke-print loop, as described in subsection 3.2. One may then consider the physical time period where the user first thinks for a while, and then types in an action expression, as a timepoint in the `Leonardo` sense. For simple, single-cycle actions the following action-period will be the period when the action gets executed, and the next timepoint will be the physical period where the next action expression is decided and typed in.

If actions extend over several cycles, then each action-period can contain timeslots for several of the actions that go on at that time. However, the 'starting time' of an action from the point of view of the system will be the last timepoint (in our specific sense of that word) before the first action-period where the action got to execute, and similarly the 'ending time' of the action will be the first timepoint after the last action-period where the action operated.

The aforesaid applies to computational actions within the computer at hand. For robotic actions and other actions that are performed outside that computer, actions can of course usually be performed with true concurrency, and the duration of an action will be defined in terms of when and how it was controlled or observed.

Other ways of using the timeline are also possible and useful. For example, in a natural-language dialog system it may be appropriate to consider each input of a spoken phrase into the system, as well as each output phrase that is produced by the system as an action in itself. In this case, `Leonardo` timepoints should characterize the starting-time and the ending-time of each input and output, instead of 'containing' such inputs. Break-ins and other situations where the user and the system perform concurrent speech acts or other communication acts can then be represented in a natural manner.

### 4.2 Computational Ramification

Since timepoints in `Leonardo` are defined in terms of the physical timeline, it is not necessary to let all timepoints and all action-periods have similar size; it is perfectly possible to let them be different even by orders of magnitude.

Consider for example the following situation. The system has decided to perform action A as a prerequisite for performing action B, where both A and B are physical actions by a robot which require nontrivial time. The stated effects of action A do not exactly match the preconditions for action B, and a few steps of logical deduction are required to infer that B can now be performed. Furthermore, these steps of logical deduction also check that some other conditions still apply and have not been invalidated while A was being performed.

The step from the immediate effects of A to the preconditions needed by B may be considered as ramifications, and in line with the usual treatment of ramifications in NRAC it would be natural to do those deductions within the last action-period of A, so that they are available in the timepoint that is the ending-time for A. However, there is also another possibility, namely considering those deductions as additional actions that take place *after* the action A has ended, albeit actions that execute very rapidly. The resulting timeline will then contain some timepoints that are wide apart, in particular the timepoints characterizing many physical actions, but it will also contain clusters of timepoints that physically occur in rapid succession, namely, timepoints that separate the inference actions.

The latter approach entails some advantages, such as the possibility of treating more or less complex inference activities as actions that can be subject to planning and other cognitive activities. It also introduces some problems, in particular the need to distinguish between cognitive (computation) time on one hand, and real-world time on the other. Anyway, it is an approach that makes computational sense, and that raises some interesting problems for the corresponding logical system.

## 5 Discussion

We proceed now to discussing the potential relevance of research on nonmonotonic reasoning, actions and change (NRAC)[2] for `Leonardo` systems, and vice versa.

### 5.1 The Relevance of Leonardo Systems for NRAC

The major reason why `Leonardo` may be relevant for reasoning about actions and change is by demanding extensions to the logic while at the same time being precise about what is required for the extension. The `Leonardo` design allows for concurrent actions, subactions, delayed effects, and others more. It also contains an explicit notion of an agent, and ways of specifying whether an agent is able and willing to perform an action requested from it. The list can be continued. These are phenomena that ought to be represented in logics of actions and change, but which are incompletely understood at present.

For these reasons, a `Leonardo` system may be seen as a precisely defined model environment for a logic of actions and change, that is, as an underlying semantics for it in the sense that was introduced in 'Features and Fluents'[4]. In that book I defined an underlying semantics and used it for assessing the range of applicability of about a dozen nonmonotonic entailment methods for such logics. This underlying semantics was a kind of simulation, expressed in set-theoretic concepts, that represents the actions of an agent in the world being modelled. However, the underlying semantics of 'Features and Fluents' does not model the sensor/actuator level in any meaningful sense. (I intended to include a chapter with those contents, but was not able to complete it in a way that I was happy with at the time the book was being written). I hope at present that the `Leonardo` language can be used for defining a more expressive and realistic underlying semantics.

The possibility of *nonmonotonic partial evaluation* is an interesting one. We described above how the update of the current state of an action is viewed as partial evaluation of its local state record. This partial evaluation is robust with respect to lack of information: it will then keep the unevaluated variable or form, and try again in the next cycle. However, it would make sense to also allow operators under which the partial evaluator is allowed to replace a form by a default value even though the information required for standard evaluation is missing. This facility ought to be of interest both for characterizing defaults due to timeout, and defaults on the level of logical reasoning.

### 5.2 The Relevance of NRAC for Leonardo Systems

The other question is as follows: if the `Leonardo` proposal is used in the design of a software system that in turn is used as a platform for various applications, will it then be possible to use existing logics of actions and change to characterize that system or its applications, or even to help in building them?

One feature of `Leonardo` is particularly important from this point of view, namely the system support for distinct

[2]Here the acronym NRAC is used for the *research area* and does not specifically refer to the NRAC as a workshop series.

timepoints whose ontology is consistent with a logic of actions and change. In this way the current state of each action is easily accessible, describable, and available for inference.

To the extent that a logic of actions and change can be made to apply to a `Leonardo` system, several uses of it come to mind:

- Use of the logic for specifying the execution of actions in effective form, so that it can be used to define the behavior of agents;
- Use of the logic for characterizing actions for the purpose of planning, and possibly for diagnosis, for example by specifying pre- and postconditions for those actions;
- Use of the logic for specifying aspects of goal-directed or social behavior, for example, when to inform another agent about some facts (and when not to do it), how to handle the failure of an action, and so forth.

The last item is a particularly interesting one since it opens up the possibility of generic services that can apply across several domains. In this context it should be an advantage to work with a general-purpose notation, as offered by `Leonardo`, rather than a specific notation that has been developed for a particular application. The simplicity of the `Leonardo` notation, which is due to it staying close to the notation of set theory, will then also be an advantage in comparison with XML and other notations that originate in the semantic web initiative and that have a much more elaborate structure.

## 6 Related Work

The key idea in `Leonardo` that is particularly relevant for the present workshop, is preparing for a synthesis between logics of actions and change, and a high-level programming language. From the language point of view, almost all the constructs in the proposed `Leonardo` language are in themselves well-known. It is not our intention to invent new and very original constructs for the language, but instead to compose selected, well-known concepts in a new way for the purpose of obtaining a basic software architecture that is powerful and exhibits as little conceptual redundancy as possible. This is necessary as a prerequisite to the 'meeting' with logics of actions and change, or in fact, with any kind of logic.

This view must however be combined with the actions-and-change point of view where the idea of integrating a logic of actions and change with a programming language is relatively unique, probably the major exception being the use of Prolog for the event calculus, by Shanahan[7] and others. `Leonardo` differs from these in a number of ways: use of several sorts including records; more general use of set theory notation for defining functions; use of expressions for marked-up text and for web-level references, and so on. The Golog[3] language also represents a step from the situation calculus, as a logic of actions and change, towards a programmable system, but it seems to be even more remote from the expressiveness of a programming language than what one finds in Prolog.

On the programming-language arena, the most strongly 'related works' are the following. First of all, the Lisp language and system, and in particular the Interlisp system[10] and the software systems of the various Lisp machines that were designed in the 1980's. Interlisp pioneered the idea of a programming environment which has then been inherited by other languages and communities, and the Lisp machines showed that it was possible to integrate operating system and programming language in a strong way.

The Smalltalk language and system[3] has integrated concepts from Lisp and Simula[4], and seems to be the strongest follower of the Interlisp design philosophy today. The Perl language[5] shows in a modern setting how the facilities that are needed on the command level of the operating system can be extended into becoming a serious altough still special-purpose programming language.

The first use of set theory for programming was to my knowledge the SETL language[6]. The style of defining functions by cases in `Leonardo`, including for the definition of recursive functions, is similar to the style in the Erlang[6] language which in turn obtained it from Prolog[7].

The definition of the top-level loop implements concurrency in a way where the current state of each concurrent action is open to inspection and can be referenced. By comparison, management of concurrency using 'detach' and 'resume' operators during an evaluation process requires complex stack management methods that are (intentionally) hidden from the program. The same applies for the use of continuations. The approach used by `Leonardo` may be perceived as more restrictive, but it is closer to the representation of current state in logics of actions and change, which is a distinct advantage from our point of view.

On another note, a number of agent systems and languages contain concepts and constructs that have been taken up in `Leonardo`. Among many, particular mention of RAPS[2] whose influence on `Leonardo` is evident.

## 7 Design History and Implementation

The core constructs in `Leonardo` have been implemented in CommonLisp, including functions for reading, printing, and evaluating `Leonardo` expressions, and for defining and executing actions. Several other facilities in `Leonardo` are similar to constructs that already exist in the DOSAR robotic dialog system that has been developed as part of the WITAS project[8], and we expect to be able to migrate them rapidly to the emerging `Leonardo` system. A description of DOSAR can be found on the website of the CASL research group[9]. Forthcoming additional articles about `Leonardo` will also be posted there.

The concepts that have been synthesized into `Leonardo` have been present in our own work during a long time and

[3]`http://www.smalltalk.org/main/`
[4]`http://www.isima.fr/asu/`
[5]`http://www.perl.org`
[6]`http://www.erlang.org/`
[7]`http://pauillac.inria.fr/~diaz/gnu-prolog/`
[8]`http://www.ida.liu.se/ext/witas/`
[9]`http://www.ida.liu.se/ext/casl/`

have evolved gradually. Many aspects of the language design, including the proposed design of agents, have also been influenced by the experience of building the robotic dialog system and its auxiliary robot simulator, as a part of the WITAS project. This applies in particular for the view of the top-level executive of the system which bears some resemblance with the robot simulator that was implemented as a tool for the development of the dialog system.

The long-term research background for the present work is documented on the CAISOR website[10].

## Acknowledgements

We gratefully acknowledge the following support. The WITAS project has been sponsored in full by a grant from the Knut and Alice Wallenberg Foundation. The concluding step of work leading to the present article was done while the author was visiting the Contraintes research group in the INRIA Research Center at Rocquencourt (Paris), France.

## References

[1] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and Jonas Kvarnström. Temporal action logics language. specification and tutorial. *Electronic Transactions on Artificial Intelligence*, 2:273–306, 1998.

[2] R. James Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*, pages 62–69, 1994.

[3] Hector J. Levesque, Raymond Reiter, Yves Lesérance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, April-June 1997.

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

[5] Erik Sandewall. Cognitive robotics logic and its metatheory: Features and fluents revisited. *Electronic Transactions on Artificial Intelligence*, 2:307–329, 1998.

[6] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[7] Murray Shanahan. *Solving the Frame Problem*. MIT Press, 1997.

[8] Murray Shanahan. A logical account of the common sense informatic situation for a mobile robot. *Electronic Transactions on Artificial Intelligence*, 2:69–104, 1998.

[9] Yoav Shoham. Reified temporal logics: Semantical and ontological considerations. In *European Conference on Artificial Intelligence*, pages 390–397, 1986.

[10] Warren Teitelman. Toward a programming laboratory. In *International Joint Conference on Artificial Intelligence*, pages 1–8, 1969.

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