-LiTH-IDA-R-88-12 ISSN-0281-4250

INSTITUTIONEN FÖR DATAVETENSKAP
UNIVERSITETET OCH TEKNISKA HÖGSKOLAN
LINKÖPING

Software Architecture and Programming Paradigms for Robotics Applications

by

Erik Sandewall

Johan Hultman

Erik Tengvald

RESEARCH REPORT
RKLLAB, May 1988

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

Computer and Information Science Department
Linköping University

PhD theses:

* Available at: University of Microfilms Intl., 300 N. Zeeb Road, Ann Arbor, MI 48106, USA.

(*Linköping Studies in Science and Technology. Dissertations.*)

*No 77	**Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982.
No 94	**Hans Lunell:** Code Generator Writing Systems, 1983.
No 97	**Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983.
*No 109	**Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984.
*No 111	**Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984.
*No 155	**Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons. 1987.
No 165	**James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning. 1987.
No 170	**Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems. 1987.

Licentiate of engineering Theses:

(*Linköping Studies in Science and Technology. Theses.*)

No 48	**Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching. 1985.
No 52	**Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems. 1985.
No 60	**Johan Fagerström:** Simulation and Evaluation of an Architecture based on Asynchronous Processes. 1985.
No 71	**Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System. 1987.
No 72	**Tony Larsson:** On the Specification and Verification of VLSI Systems. 1986.
No 73	**Ola Strömfors:** A Structure Editor for Documents and Programs. 1986.
No 74	**Christos Levcopoulos:** New Results about the Approximation Behaviour of the Greedy Triangulation. 1986.
No 104	**Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology. 1987.
No 108	**Rober Bilos:** Incremental Scanning and Token-based Editing. 1987
No 111	**Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments. 1987.
No 113	**Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge. 1987.
No 118	**Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs. 1987
No 126	**Dan Strömberg:** Transfer and Distribution of Application Programs. 1987
No 127	**Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems. 1987
No 139	**Christer Bäckström:** Reasoning about Interdependent Actions. 1988.
No 140	**Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing. 1988.

LiTH-IDA-R-88-12 ISSN-0281-4250

Software Architecture and Programming Paradigms for Robotics Applications

by

Erik Sandewall

Johan Hultman

Erik Tengvald

Abstract: We propose a layered software architecture intended for robotics applications, where there is a data flow from sensors to actuators.

This work was made possible by the support of and cooperation with the following organizations:

STU (ramprogram, Prometheus)
FOA
FMV
SUTEC AB

Postadress:
Institutionen för datavetenskap
Universitetet i Linköping och
Tekniska Högskolan
581 83 Linköping

Mailing address:
Department of Computer and
Information Science
Linköping University
S-581 83 Linköping, Sweden

1. Software architecture vs. programming paradigms.

We use the term robotics in the sense of 'the intelligent connection from sensors to actuators' [Lozano-Perez, 1986], and study design principles for robotics software. This article analyzes two interdependent issues, namely the software architecture used in robotics systems, and the programming paradigms which would be appropriate for implementing such systems. We first describe the architecture, and then generalize it into a programming paradigm. We also discuss or touch upon the following specific aspects of the architecture, and how they can be implemented in the paradigm:

temporal bindings of measurement data
dependency nets and reason maintenance
operational states
entry-point vs. exit-point controls of data flow
incremental update of 'compiled' action plans

The architecture of robotics software systems has been extensively studied in the contexts of intelligent autonomous vehicles for terrain, under-water, or other environments, and in the context of automatic manufacturing systems with built-in intelligence. The general character of such systems is usually described like in figure 1. There is a need for a "low level" data flow from sensors to actuators, for immediate control of the movements or maneuvers of the robot. There is also a need for several "higher levels" where sensory data is interpreted, used as one of the bases for planning and plan revision, and then indirectly affects the robot's actuators.

It is commonly recognized that artificial intelligence techniques must be used for some parts of such a control system, and particularly for the higher levels of processing. At the same time it is clear that the lowest level(s) of data flow can not be adequately handled with the usual A.I. software methods. Almost all systems of this kind therefore make use of conventional software techniques (real time operating systems, programming languages, etc.) for implementing the lowest system levels.

Some authors (Brooks, 1986; Rosenschein, 1987) make use of a special 'dataflow' paradigm for characterizing the lower system layers. The processing there is characterized by a data flow diagram, consisting of nodes and arcs, where successive (over time) values of sensory data travel along the arcs in the diagram, and the nodes represent various transformations on the data elements - arithmetic operations, integration, conditional choice, etc.

For Rosenschein, the primary reason for using the dataflow paradigm is to define the structure of the lower layers sufficiently well so that its operations can be effectively controlled by higher layers. Brooks emphasizes rather the software engineering aspects of the same paradigm: it supports the orderly manual development of the robot's software from the bottom up. We believe that both those aspects are important, and the programming paradigm described in this paper is intended to serve both purposes.

The alternative to introducing the separate paradigm is to build a hybrid system where lower levels are implemented directly in a conventional programming language, and the higher levels using standard A.I. techniques. Either of two problems may then arise. If only minimal services are provided in the conventional software environment, and as much functionality as possible is allocated in A.I. environments based on e.g. Lisp or Prolog, then it is not possible to achieve adequate performance at reasonable cost. On the other hand, if non-trivial low level services are "hard coded" in the conventional software environment, then it may become very difficult for the higher system levels to control or influence the hard coded ones. The points where converging arrows meet, in figure 1, are the points where difficulties arise.

In order to avoid that problem, one must either find a programming paradigm which can be used homogenously on all levels of the system, or define the lower layers in such a way that the higher layers can "understand" them, or both. This observation defines our task: we must both formulate the appropriate software architecture, and find a programming paradigm to match it.

2. The software architecture.

The general design of the arcitecture is described by figure 2, and follows the general pattern of figure 1. The system has three layers, called the *process layer, rule layer*, and *analysis layer*, respectively. Each layer has a *local state* which is a function of time, and is updated at regular intervals. The period from one state update to the next is called a *phase* for the layer. Lower layers operate with shorter phases, i.e. update their parameters with higher frequency, as usual in automatic control systems.

The rules for calculating the new layer state from the previous one are expressed using a data-flow diagram, as illustrated in figure 3. Each wire represents a parameter; each box, an operation that receives input

parameter(s) and produces output parameter(s). The lowest, process layer implements the immediate connection from sensors to actuators, and low level analysis of sensor data for recognition purposes. Therefore the local state of the process layer is typically an ordered set of numerical parameters, and the data-flow diagram implements the feed-back control loops.

We wish the architecture to include also the tight feedback loops for sensory-motoric control. When real-life applications with high speed behavior are addressed, using standard techniques for automatic control, it is necessary to use constant phase lengths. We therefore do not use the wholly asynchronous architecture proposed by Brooks.

The analysis layer at the top of the hierarchy has a local state as described in figure 4, consisting of a current world model, a goal model, and an action structure. The current world model and the goal model have the same structure, and differ only since one refers to the present time, and the other to the intended future. The action structure is a set of action descriptions, together with a partial order on the time-points (starting times and ending times) of those actions. Notice in particular that the local state of the analysis layer contains descriptions of several states of the world.

The rule layer (middle layer) is the vehicle for implementing the actions in the action structure, in terms of the data-flow on the process level. For each action in the action structure, there is a corresponding action-active parameter in the local state of the rule level which signals whether the action is currently active (T) or not (F). For each observed criterium which indicates whether an action should start or stop, there is also a parameter in the same local state. The data-flow diagram within the rule layer represents rules which determine some of those parameters from others, for example which turn on and turn off the action-active parameters on the appropriate conditions. Those rules are derived from the action structure in the analysis layer. More details about the representation of action structures in the rule layer are given in Hultman and Nyberg, 1988.

The setting of the parameters in the rule layer is made available also to the process level, in ways which will be described in more detail below.

The rule layer is not entirely derived from the action plan, however. There must also be effective ways whereby the system can respond promptly to emergencies or other unforeseen situations. The rule layer also contains devices

for this purpose, allowing it for example to switch instantly to an alternative behavior pattern (such as "hide" or "run away") in the case of a danger, and to postpone until later on, the process of reconsidering the situation and making a new action plan. The two sources for the rules in the rule layer are indicated by figure 5.

The system is explicitly intended to be workable using only the lower two layers, process layer and rule layer. In this case a "dumb" system is obtained, and it has to be programmed manually. For many practical applications of autonomous robots this may however be quite sufficient. The analysis layer is an extra facility which may be added for providing the system with a certain measure of intelligence. Still all three layers have been designed together, and the lower two layers have been explicitly chosen in such a way as to facilitate the job for the analysis layer whenever it is needed.

This paper concentrates on the more detailed aspects of the process and rule layers, and deals more briefly with the analysis layer. Other papers from our group focus on the analysis layer and on specific applications. In this paper we first consider some important reasons for, and aspects of the software architecture, and then describe the more general programming paradigm and computing 'engine' wherein this design can conveniently be implemented.

3. On the need to separate data flow and control flow in systematic ways.

It is well understood that conventional programming languages integrate control flow and data flow by making control flow the primary structuring factor in the program. Block structure, loop constructs, and procedures are some of the conceptual devices used for expressing control flow. A significant part of the work in understanding a program is used for reconstructing the data flow.

Message-passing and other data-flow programming languages attempt to reverse the roles of these two types of flow, by making data flow primary. However data-flow alone does not solve the problem outlined above.

The "plan calculus" of Rich et al. (see e.g. [Rich 1981]) is a notation which separates data flow and control flow on an equal footing. Experiments have been made with automatic translation from expressions in the plan calculus, to

programs in a conventional programming language.

In our paradigm, data flow and control flow are kept separate not only at design time, but also at execution time. The data flow is used within the lowest level; the control flow is used from the next higher level for directing the operations of the lowest level.

For example, consider the operation of moving a robotic vehicle up to a wall, and stopping there. An encoding of this operation as a conventional procedure will receive and interpret sensory data, control the actuators, and accomplish the movement. Also the same procedure will check when the wall has been reached, terminate the procedure at that time, and report success.

By contrast, if data flow and control flow are separated, there will be one module for driving towards the wall, and another module for recognizing arrival. Both modules will continously report their performance and observations to the next higher level in the system hierarchy. When the higher level decides that no more driving is needed, it will tell the drive-towards- wall procedure to terminate. One of the advantages of this alternative principle is that different possible reasons for stopping the driving process, including successful termination, vehicle failures, obstacles, and parenthetical interrupts, will be handled by the same system level.

An important point is that processes will now be required to return progress information continously to their next higher level, rather than just returning a report of success (or failure) when they are finished. This is analogous to how a person leads other people working for him or her.

4. The programming paradigm.

We now proceed to an outline of a programming paradigm for robotics applications (PARA) which generalizes the principles that were discussed in previous sections.

A PARA 'program' operates on a set of parameters. In the simplest case the parameters have no grouping, and can be seen as a simple vector. In general, however, the parameters are structured hierarchically as suggested in figure 6, into a tree structure consisting of several branches. Each branch is an ordered set of parameters.

Parameters are updated with a constant frequency; parameters in the same branch are updated with the same frequency. If there is more than one branch, then the frequency of a lower branch is an integer multiple of its superordinate's frequency. Parameter values may be transported up and down between a superordinate and a subordinate branch, but not directly between two subordinates.

The time period from one parameter update to the next is called a phase. For each branch, the transition during a phase is defined by a data-flow diagram. Figure 7b illustrates how the two-branch structure in figure 7a is updated using one data-flow graph for the higher level, and another data-flow graph which is used with higher frequency for the lower level.

In programming language terms, this means that we have a single- assignment language within each phase, but with parallel assignment at the end of the phase. For example, the structure shown in figure 8 will swap the values of the two parameters. In general, first all the new values for all parameters are calculated, and then all the new values are stored.

For communication between levels, there need to be "downward" parameters which are written by the superordinate branch and read by the subordinate branch, and other "upward" parameters which go the other way. For example, a feedback loop of classical kind can be seen as a branch with relatively high frequency, where the control value is determined by the superordinate branch and communicated using a downward parameter. Also a subordinate branch may integrate the value of a parameter during those successive phases of that branch, which fall into one phase of the superordinate branch, and report the integrated value back to the superordinate using an upward parameter.

Consider now the action structure that was described in figure 4 above. We propose to represent it as follows. A higher parameter branch contains the action structure, coded (at least) with one boolean parameter for each action in the structure. The parameter is T if the action is active, and F otherwise. A lower branch contains dependencies that implement all the actions. The very simple action structure shown in figure 9a, has the realization shown in figure 9b. The actuator M should receive a value which is calculated in either of two ways, depending on whether action A or B is active. This is realized by two operations on the data flow graph. The first operation, labeled 1 in the figure, outputs the symbol A if the value of parameters A and B are (T,F), symbol B if the values are (F,T), and causes an error if the values are (T,T). The other

operation, labeled 2, gives an output which is equal to the first or the second input, depending on whether the 'control line' (from operation 1) contains an A or a B.

The upper layer must therefore be able to determine the appropriate values for parameters A and B, based on other parameters that express sensory information. The parameter dependencies on that level may consist largely of rules, of the same kind as in rule-based systems, but with continous update.

The PARA paradigm is clearly well suited for specifying and implementing the process and rule layers of the architecture described in section 2. The option of a tree structure with several branches, and different update frequency in different branches, makes it possible to see even low-level and high-speed control loops as parts of the same paradigm. It also readily suggests a realization in a distributed environment, where the operating system is equipped to handle the branches of the parameter tree.

The same paradigm can also be readily extended upwards, for use in the analysis layer of the architecture. Some straight-forward extensions are required: parameters must be allowed to take symbols and list structures as values; operations in the data- flow diagrams must include list operations such as the familiar car, cdr, and cons.

5. Temporal binding of data.

In this section we return to the issue of synchronous vs. asynchronous operation of the data-flow, especially in the process layer. In any system where there is a data flow from sensors to actuators, there arises the question of how to handle the time delays. What time does the arriving actuator data refer to, and which part of the system takes responsibility that they are applied at the right time?

If conventional software engineering methods are used, these considerations are 'hard-wired' into the manually written programs. In a flexible architecture, where the data flow diagram is generated and modified from higher system layers, we can see two possible principles:

a) Asynchronous operation with time-stamped data. As a system- wide convention, all data are tagged with the time-point they refer to. Each 'pipe' in

the data flow transmit a sequence of (time,value) pairs. Each logical sensor generates a flow of such pairs, where the time specifies when the value was measured. Each logical actuator also receives a flow of such pairs, where the time component in a pair should normally be $\geq$ current real time, and applies the specified value at the specified time. "Type-ahead" on the actuators is possible. The operations along the data flow line use one or more (t,v) pairs for calculating a ($t+\Delta t$,v') value for the actuator. Our COPPS system (Hultman, 1987) uses this method.

b) Synchronous operation with constant frequency for each sub- system. This is the method which is used in the presently described architecture and paradigm.

The reason why we prefer synchronous operation is that it better supports conventional automatic control techniques, which are of importance for guaranteeing rapid and stable behaviors. Also universal time-stamping of data creates a performance overhead which is not always acceptable nor necessary, forcing some parts of the system out of the chosen architecture. Finally for those parameters where time-stamping is necessary, it can readily be implemented in the synchronously operating PARA paradigm as follows. Current time is considered as a parameter in each branch of the parameter tree, and sensor data are time-stamped by 'consing' (pairing) them with the current time parameter and then sent onwards through the data flow. At the receiving end, logical actuators receive time-stamped value pairs and delay their use until real time equals the time-stamp.

6. Dependency nets and reason maintenance.

In principle, the PARA paradigm calls for the whole data-flow diagram to be re-executed during each phase, so that each parameter is re-evaluated. In practice, many parameters may remain constant from one phase to the next, so that many operations in the diagram evaluate to the same value as in the last cycle. This may particularly often be the case in the rule layer, where many parameters have boolean or other discrete values. It is then tempting to reduce the computational load by keeping track of where changes have occurred, and only update those parts of the diagram where it is necessary. In this way the rule layer obtains the character of a reason maintenance system.

There are also obvious good reasons for allowing parameters to take the value 'undefined' or 'unknown', and to allow operations in the data flow diagram which output a default value if the (an) input is unknown. The techniques for doing this are well known from work on reason maintenance systems.

There is however a significant problem in this context, namely the danger of computational overload when some changes do happen, for example in an emergency. We propose to deal with this problem using the techniques of operational states and derived entry-point controls, which are the topics of the following sections.

7. Operational states.

The classical model of an intelligent robotic system goes as follows: the system receives directives of what to accomplish. A knowledge-based planner calculates a plan, e.g. an action structure as used in our analysis layer. The actions in the plan are executed. If something unexpected occurs during plan execution, the system halts, replans, and resumes operation.

The problem with that model is that many interrupts require rapid reactions, and can not wait for an elaborate re-planning process. This problem can not be dealt with by "incremental planning" where the old plan is revised rather than building a new plan from scratch. Incremental planning with full generality will also be slow.

We propose to handle interrupts as follows. A number of distinct operational states are defined for the system. Examples of operational states for an autonomous underwater vehicle might be:

rest
scan an area
go to a target
hide
evade hostile vehicle(s)
limp (vehicle damaged)
manual control

Examples of operational states for a driver-assistance system in an automobile (as studied in the Prometheus project) might be:

driver control

cruise in lane on motorway
driver incapacitated - emergency
obstacle ahead - emergency
dense city traffic - accept directives of urban traffic control system

Instead of a mapping from actions to data-flow procedures, we have a mapping from

actions x operational-states

to data-flow procedures, i.e. for each combination of action and operational state there is such a procedure. (By data-flow procedure we mean a segment or part of the data-flow diagram). By appropriate mechanisms the system may be induced to shift its operational state. For the currently executing actions, this has an effect like throwing a big, multi-polar switch: all the actions instantly change their definitions.

Operational states are important not only for the process layer, but also for dealing with the danger of overload in a rule layer which operates on a reason maintenance basis. We can design the rules so that in some (emergency) operational states, only a subset of the rules are active. If the system senses that overload occurs, it switches to such an emergency state in order to off-load itself.

It seems appropriate that the phase length, i.e. update frequency may be different in different operational states. This may be used in emergency states, both for achieving very rapid actions in the process layer, and on the other hand for allowing more update time in the rule layer when more changes occur.

8. Entry-point vs. exit-point control.

The major use of operational states is that current activities are determined by a combination of the action-active parameters and the current operational state. The most obvious way of realizing operational states in the PARA paradigm is to let one of the rule layer parameters have the current operational state as its value, and to also make that parameter downward accessible for the process layer. The rule system on the rule level can then assign current operational state in the same way as it assigns all its other parameters.

This solution means that the logical control from the rule level to the process level is exerted by the combination of the operational state parameter and the action-active parameters. What about the risk that the process level will

become too complicated as a result?

We propose to reduce that risk by the following method. Suppose the data-flow diagram on the process level contains several parallel segments which calculate alternative values for a certain actuator controlling parameter (figure 10). Only one of them will be active at any one time, and one could therefore have entry-point controls which allow data to flow into the presently active parts of the diagram, and not into the inactive parts.

The alternative method is to only use exit-point controls, i.e. to allow data to flow into all the alternative segments, but to only use one of the resulting values. The arrangement shown in figure 9 was an example of exit-point control.

Exit-point controls are potentially wasteful of system performance. However they have the advantage of being more transparent, and in cases where the choice between the alternatives is determined in complicated ways depending on both current-state and action-active parameters, exit-point controls may be necessary for retaining the system's transparency.

One can have the best of both worlds by calculating the entry-point constraints automatically (as far as possible) from a description of the exit-point controls and their logical relationships. In this way it should be possible to reduce the performance price that has to be paid for the convenience of exit-point controls. This technique is also appropriate when operational states are used to reduce the computational load in the rule layer.

9. Functional and logical aspects of the analysis layer.

The local states of the process and rule layers could be literally seen as parameter vectors. For the analysis layer a more versatile point of view is needed.

In some ways the local state is like a 'blackboard' in the A.I. sense, since it contains data that is publicly available within the layer and can be updated freely. The real-time constraints and the synchronous update however differentiate it from conventional blackboards.

From a logical perspective, we view the system state as a partial interpretation, i.e. as a mapping which assigns values to function symbols and relation symbols of zero or more arguments. The lowest layers maybe only use constant symbols (zero-argument function symbols) and proposition symbols (zero- argument relation symbols), but higher system layers will certainly find uses for the more general case.

While the system state is seen as an interpretation in the sense of logic, we must admit it to be a partial and not necessarily a total interpretation, since the values of some parameters may be unknown. This introduces the possibility and the need for default reasoning, as a theoretical basis also for the reason-maintenance view of the rule layer.

We have previously made theoretical studies of action structures, where each action is characterized by a pre-condition, a post-condition, and a prevail condition, and each of these three conditions is a partial interpretation (Sandewall and Rönnquist, 1986; Sandewall, 1987). Recently Bäckström has introduced also a fourth 'keep' condition into the theory, in order to characterize interdependence between parallel actions (Bäckström 1988). These theoretical results are immediately applicable to the software structure described here, where the data-flow diagrams represent transitions from one parameter vector (partial interpretation) to another.

The lower levels are assumed to run on a regular frequency, i.e. with a constant phase length. It is not yet clear to us whether higher levels should also follow this pattern, or be allowed to run with variable phase length. An extension of the PARA paradigm in the direction of asynchronous operation may be necessary.

10. Compiling action plans into rules in the rule layer.

The relationship between the analysis layer and the lower layers is one of incremental compilation. When everything is normal, the lower layers contain information which has been 'compiled' from the current state of the analysis layer, and which when executed in the correct way should lead to the goals eventually being achieved. The analysis layer only has to intervene in the lower layers when things do not go as expected, and its intervention may then be to edit the data-flow diagrams in both the rule layer and the process layer. Conversely, the sensory information that the analysis layer requests the lower

layers to send upwards, is whatever the analysis layer needs for checking the progress of the planned action structure. The detailed presentation of this topic is however outside the scope of the present paper.

11. Previous work.

The results described in this paper have evolved in the usual design-iteration process, meaning that a design is followed by an implementation, in the course of which it is realized what the design should have been like, and a new design develops.

The COPPS system (Hultman, 1987) preceded the PARA design and provided a large part of the method. It is a prototype intelligent control system originally developed for controlling (models of) machine groups in automatic manufacturing. COPPS was implemented in 1986 and 1987. The major change from COPPS to the PARA paradigm is that in COPPS the action structure was closely integrated with the corresponding segments of the data-flow diagrams. The new design is more general, and in particular makes it easier to handle operational states and interrupts.

At the same time, one of us (Tengvald, 1987) has developed the HIDESHAPE programming language, which is a very general and powerful paradigm for parallel symbolic computation. The paradigm proposed in this paper relies heavily on HIDESHAPE, but has been simplified and adapted for the needs of robotics applications. From the present paper's point of view, HIDESHAPE demonstrates how the PARA paradigm can be extended into a full- fledged programming system. This is of particular importance for the topmost, analysis layer in the architecture.

References.

Brooks, 1986. R. Brooks: A robust layered control system for a mobile robot. IEEE Journal of robotics and automation, Vol. 2, Nr. 1, 1986.

Bäckström, 1988. Ch. Bäckström: Keeping and forcing. How to represent cooperating actions. RKLLAB Memo, 1988.

Hultman, 1987. J. Hultman: COPPS, A software system for defining and controlling actions in a mechanical system. Proc. IEEE workshop on languages for automation, Wien, 1987.

Hultman, 1988. J. Hultman and A. Nyberg: Realisering av handlingsplaner och responsregler i systemverktyget. RKLLAB Memo, 1988.

Rich, 1981. Ch. Rich: A formal representation for plans in the programmer's apprentice. Proc. IJCAI 1981.

Rosenschein, 1987. S. Rosenschein: Formal Theories of Knowledge in A.I. and Robotics. Report CSLI-87-84, CSLI, Stanford, CA; and personal communication.

Sandewall, 1986. E. Sandewall and R. Rönnquist: A representation of action structures. Proc. AAAI 1986.

Sandewall, 1987. E. Sandewall: The pipelining transformation on plans for manufacturing cells with robots. Proc. IJCAI 1987.

Tengvald, 1987. E. Tengvald: The AIM project: establishing a knowledge engineering environment for the application of artificial intelligence techniques in the manufacturing industry. Symposium "A.I. - supercomputers", Umeå, Sweden, 1987.

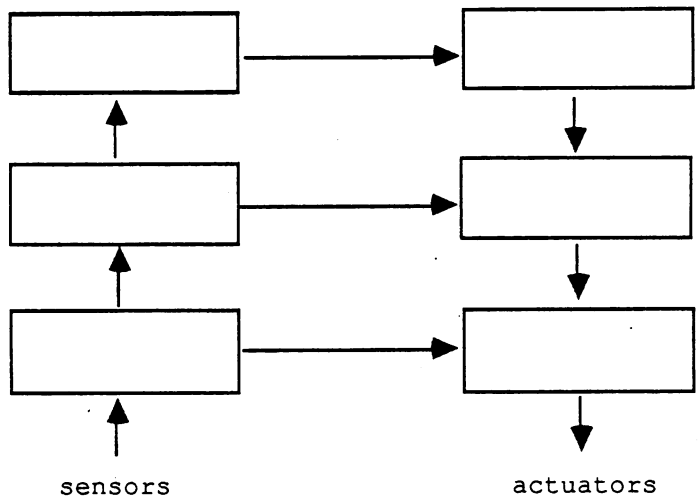

Figure 1.

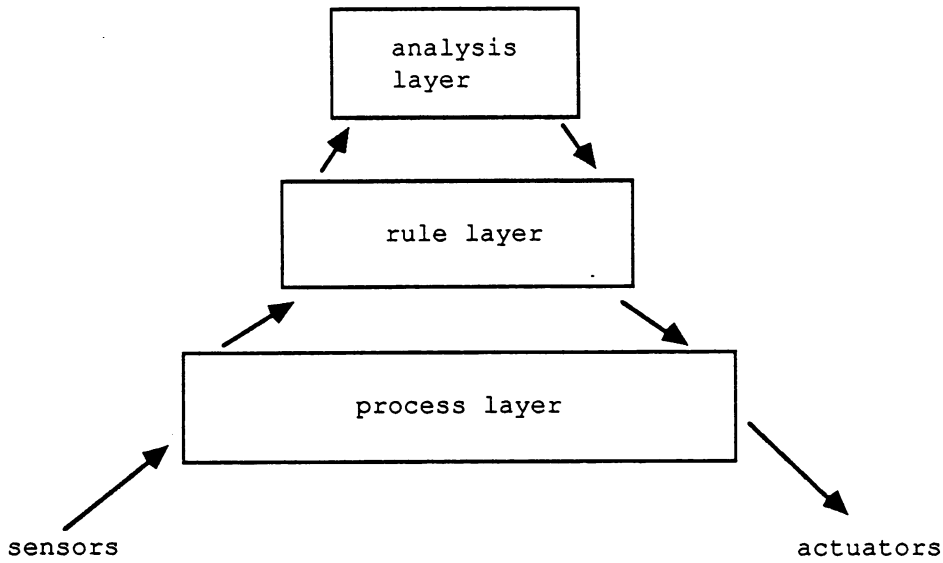

Figure 2.

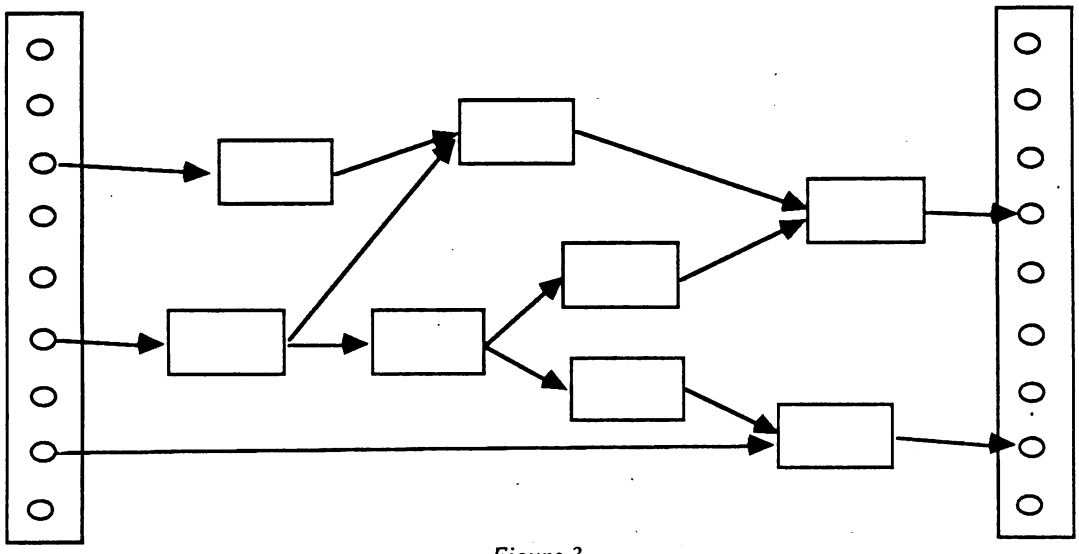

Figure 3.

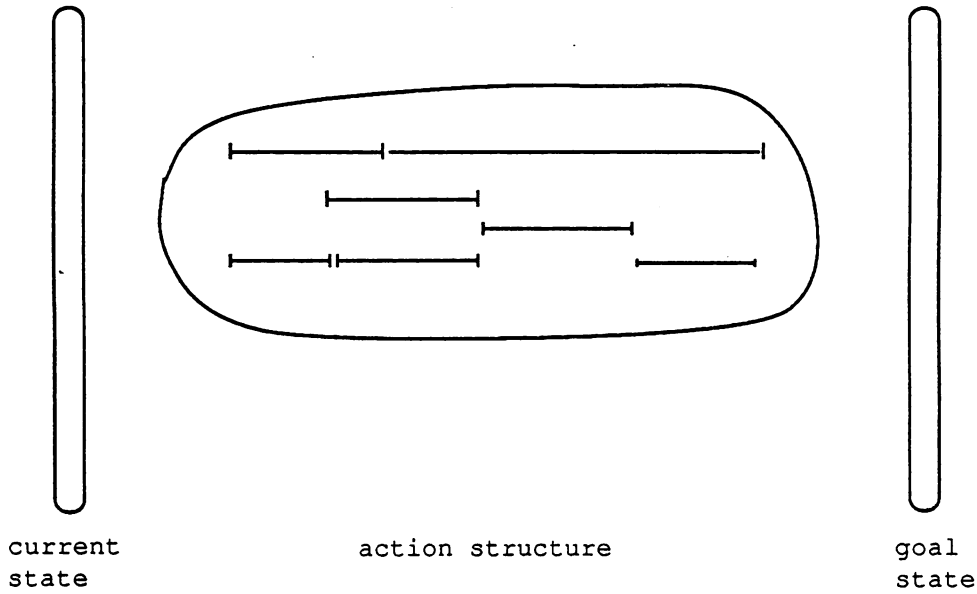

Figure 4.

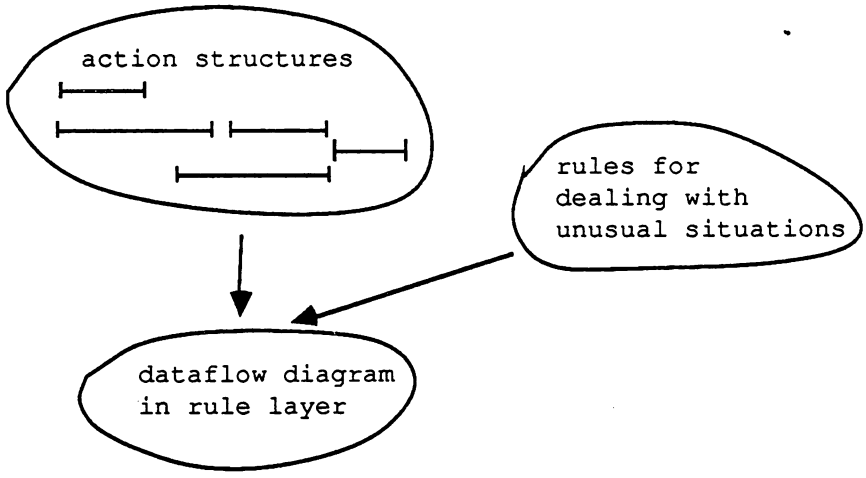

Figure 5.

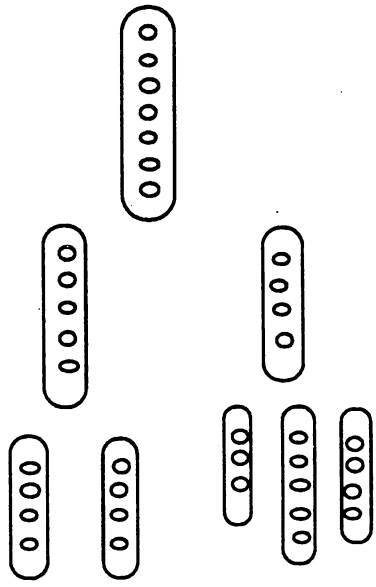

Figure 6.

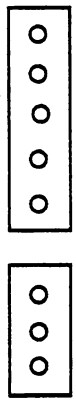

Figure 7a.

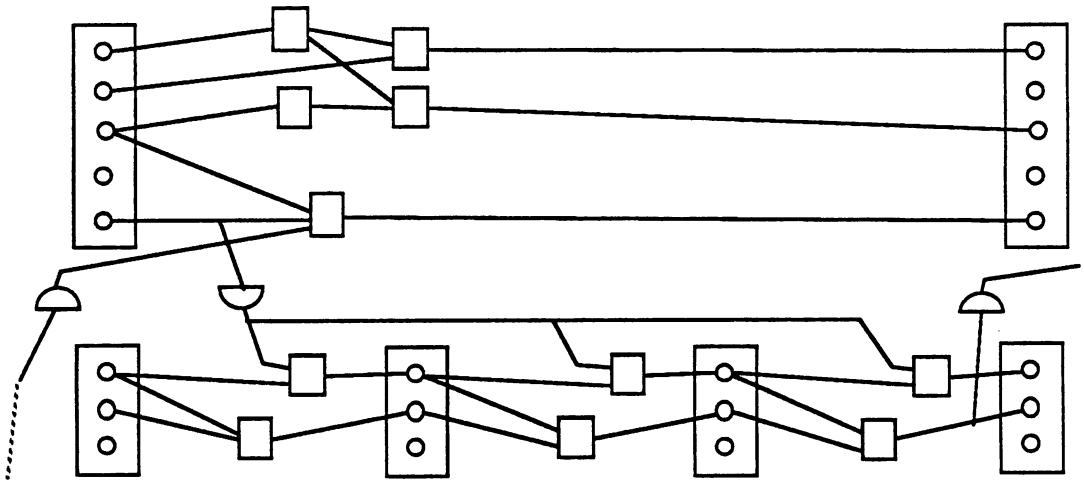

Figure 7b.

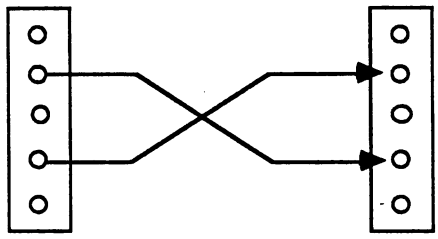

Figure 8.

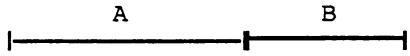

Figure 9a.

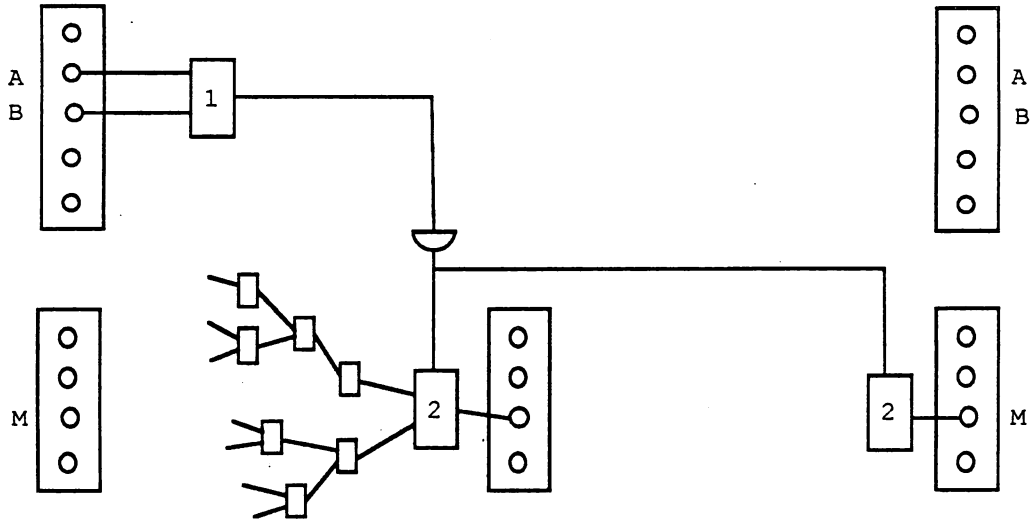

Figure 9b.

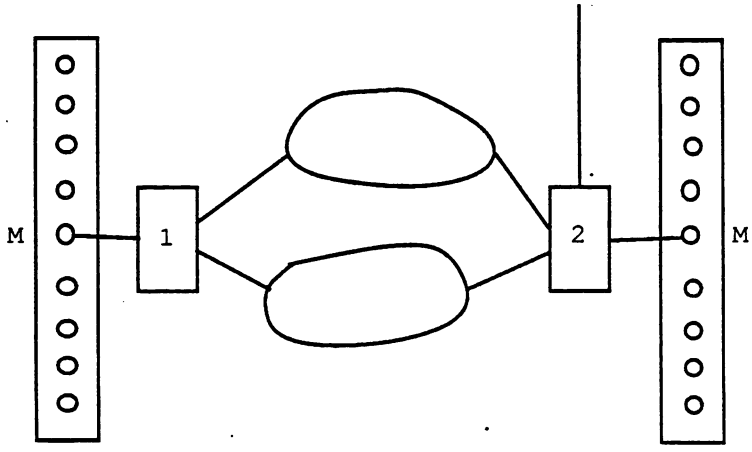

Figure 10. 1 = entry point control, 2 = exit point control.

Avdelning, institution, fakultet
Division, department, faculty

Department of Computer and
Information Science
Linköping University

ISBN:

ISSN: ISSN-0281-4250

Rapportnr:
Report no: LiTH-IDA-R-88-12

Upplagans storlek:
Number of copies:

Datum:
Date: May 1988

Projekt:
Project:

Titel:
Title: Software Architecture and Programming Paradigms for Robotics Applications

Författare:
Author: Erik Sandewall, Johan Hultman and Erik Tengvald

Uppdragsgivare:
Commissioned by:

Dnr.:
Call no:

Rapporttyp:
Kind of report:

- ☐ Examensarbete/Final project
- ☐ Delrapport/Progress report
- ☐ Reserapport/Travel report
- ☐ Slutrapport/Final report
- ☒ Övrig rapport/Other kind of report

Rapportspråk:
Language:

- ☐ Svenska/Swedish
- ☒ Engelska/English
- ☐ _____

Sammanfattning (högst 150 ord):
Abstract (150 words):

Abstract: We propose a layered software architecture intended for robotics applications, where there is a data flow from sensors to actuators.

Nyckelord (högst 8):
Keywords (8):

Bibliotekets anteckningar

A Selection of Previous Research Reports.

LiTH-IDA-R-88-04 **Wlodek Drabent, Simin Nadjm-Tehrani, Jan Maluszynski:** Algorithmic Debugging with Assertions.

LiTH-IDA-R-88-03 **Staffan Bonnier, Jan Maluszynski:** Towards a Clean Amalgamation of Logic Programs with External Procedures. Accepted to the *Joint Fifth Int. Logic Programming Conf. and Fifth Logic Programming Symp.*, Seattle Washington, August 15-19, 1988.

LiTH-IDA-R-88-02 **Ulf Nilsson:** Inferring Restricted AND-Parallelism in Logic Programs using Abstract Interpretation.

LiTH-IDA-R-88-01 **Jacek Leszczylowski, Staffan Bonnier, Jan Maluszynski:** Logic Programming with External Procedures: Introducing S-Unification. A shorter version is to appear in *Information Processing Letters*, North Holland.

LiTH-IDA-R-87-26 **Jonas Löwgren:** Applying a Rapid Prototyping System to Control Panel Dialgues.

LiTH-IDA-R-87-24 **Sven Moen:** Drawing Dynamic Trees.

LiTH-IDA-R-87-22 **Lars Ahrenberg, Arne Jönsson:** An Interactive System for Tagging Dialogues. Also in *Proc. of 14th Conference of the Association for Literary and Linguistic Computing*, Göteborg 1-5 June, 1987.

LiTH-IDA-R-87-21 **Harold W. Lawson, Jr.:** Challenges and Directions in Computers and Education.

LiTH-IDA-R-87-20 **Krzysztof Kuchcinski, Zebo Peng:** Parallelism Extraction from Sequential Programs for VLSI Applications. This paper is to appear in *Microprocessing and Microprogramming, the Euromicro Journal*, 1988.

LiTH-IDA-R-87-19 **Zebo Peng:** Construction of Asynchronous Concurrent Systems from their Behavioral Specifications. Also in *Proc. of the 10th World Computer Congress*, Dublin, Ireland, Sept. 1-5, 1986.

LiTH-IDA-R-87-18 **Henrik Nordin:** Reuse and Maintenance Techniques in Knowledge-Based Systems.

LiTH-IDA-R-87-17 **Tony Larsson:** Specification and Verification of VLSI Systems Actional Behaviour This is a close version of a paper presented at *the 8th international conference on Computer Hardware Description Languages, CHDL,* 87.

LiTH-IDA-R-87-16 **Shamsul I. Chowdhury:** State of the Art in Statistical Expert Systems. Also in *Proc. of the Conference on Expert Systems and their Applications*, Avignon, May 1987.

LiTH-IDA-R-87-15 **Nils Dahlbäck:** Kommunikation med datorer i naturligt språk -' vad är det och vem behöver det?

LiTH-IDA-R-87-14 **Peter Fritzson:** Window System Architectures - an Overview. Also presented at *A Tutorial on Software Development Environments,* Linköping University, Nov 24-25 1986.

LiTH-IDA-R-87-13 **Mats Wirén:** A Comparison of Rule-Invocation Strategies in Context-Free Chart Parsing. Also in *Proc. of the Third Conference of the European Chapter of the Association for Computational Linguistics,* Copenhagen, April 1—3, 1987.

LiTH-IDA-R-87-11 **Sture Hägglund, Christer Hansson, Tomas Sokolnicki:** Knowledge-Based Training of Case Management Routines and Emergency Procedures. Also in *Proc. of the 3rd Int. Conf. on Expert Systems*, London, 1987.

LiTH-IDA-R-87-10 **Andrzej Lingas:** On Parallel Complexity of the Subgraph Isomorphism Problem.

LiTH-IDA-R-87-09 **Andrzej Lingas, Marek Karpinski:** Subtree Isomorphism and Bipartite Perfect Matching are Mutually NC Reducible.

LiTH-IDA-R-87-08 **Christos Levcopoulos, Andrzej Lingas, Jörg-R. Sack:** Nearly Optimal Heuristics for Binary Search Trees with Geometric Generalizations. Also in *Proc of ICALP'87*, Karlsruhe, July 1987.

LiTH-IDA-R-87-07 **Peter Haneclou:** A Formal Approach to Reason-maintenance Based on Abstract Domains.

The Department of Computer and Information Science
Linköping University

organizes undergraduate and graduate studies in *Computer Science*, *Telecommunication and Computer Systems*, and *Administrative Data Processing*. Research activities have an emphasis on advanced software technology and computer systems design and are organized in a number of research laboratories:

- *ACTLAB - Laboratory for Complexity of Algorithms*, which is concerned with the design and analysis of efficient sequential and parallel algorithms, and complexity theory, especially in the areas of computational geometry, data structures on bounded domains and graph algorithms.
- *AIELAB - Artificial Intelligence Environments Laboratory,* which conducts research on representation and manipulation of knowledge, problem solving techniques and expert systems with mechanical engineering applications.
- *ASLAB - Application Systems Laboratory,* which studies design of advanced support systems for interactive use of computers, including tools for automated construction of applications software.
- *CADLAB - Laboratory for Computer-Aided Design of Electronics*, which concentrates its research activities around tools for integrated development of hardware and software, graphics-based modelling and simulation techniques.
- *LIBLAB - Laboratory for Library and Information Science,* which studies methods for access to documents and the information contained in the documents, concentrating on catalogs and bibliographic representations, and on the human factors of library use.
- *LOGPRO - Laboratory for Logic Programming*, which concentrates its research activities around foundations of logic programming, relations to other programming paradigms and methodology.
- *NLPLAB - Natural Language Processing Laboratory*, which conducts research related to the development and use of natural language interfaces to computer software.
- *PELAB - Programming Environments Laboratory*, which works with design of tools for software development, specific functions in such tools and theoretical aspects of programs under construction.
- *RKLLAB - Laboratory for Representation of Knowledge in Logic*, which covers issues and techniques such as non-monotonic logic, probabilistic logic, modal logic and truth maintenance algorithms and systems.

Research Reports 1988

LiTH-IDA-R-88-12	**Erik Sandewall, Johan Hultman, Erik Tengvald:** Software Architecture and Programming Paradigms for Robotics Applications.
LiTH-IDA-R-88-11	**Mats Wirén:** An Incremental Chart Parser for PATR.
LiTH-IDA-R-88-10	**Mats Wirén:** A Control-Strategy-Independent Parser for PATR. Revised version of paper appearing in *Proc. of First Scandinavian Conf. on Artificial Intelligence*, Tromsö, Norway, March 9 - 11, 1988.
LiTH-IDA-R-88-09	**Peter Fritzson:** Incremental Symbol Processing.
LiTH-IDA-R-88-08	**Erik Sandewall:** Formal Semantics for Reasoning about Change with Ramified Causal Minimization.
LiTH-IDA-R-88-07	**Erik Sandewall et al:** IDA Annual Report 1987.
LiTH-IDA-R-88-06	**Christer Bäckström:** A Representation of Coordinated Actions Characterized by Interval Valued Conditions.
LiTH-IDA-R-88-05	**Christer Bäckström:** Keeping and Forcing: How to Represent Co[illegible]