# An AI-based Design of a Driver Information Unit

Jacek Malec
Magnus Morin*
Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden
tel. +46 13 282362   +46 13 282425
fax +46 13 282606
jam@ida.liu.se   mmo@ida.liu.se

May 13, 1996

**Abstract**

The paper describes an Artificial Intelligence-based approach to the design of a driver information unit (DIU). As the amount of information to be processed by a module presenting the information to the driver is huge, and conventional design methods do not sufficiently take into account driver workload and timing problems, it seems appropriate to use artificial intelligence techniques when designing such a system. A three-layered software architecture and a set of layer-specific software engines are used to facilitate the design process. The paper focuses on presenting the software architecture, the software tools developed and their applicability to the design of Driver Information Unit.

## 1 Introduction

The paper describes an Artificial Intelligence-based approach to the design of a driver information unit (DIU). As the amount of information to be processed by a module presenting the information to the driver is huge, and conventional design methods do not sufficiently take into account driver workload and timing problems, it seems appropriate to use artificial intelligence techniques when

*This research was supported by the Driver Assistance and Local Traffic Management project within the Swedish RTI research program '91–'94. Main sponsors of this project are The Swedish National Board for Industrial and Technical Development, AB Volvo, Saab-Scania AB and The Swedish National Road Administration.

designing such a system. We have previously defined a three-layered architecture [16] suitable for implementing real-time systems possessing a considerable amount of autonomy and intelligence. The architecture consists of:

- the *process layer* used for direct control of effectors and output devices. It is realized as a (possibly distributed) computer system equipped with real-time operating system and appropriate software for handling continuous flow of information from and to the environment;

- the *discrete response layer* (DRL) used for processing symbolic, discrete information originating in the process layer. A symbolic characterization of environment condition is created in the process layer, and every change of this characterization causes a message to be sent to the DRL. This layer makes then a decision whether control modes implemented in the process layer should be changed and how this should be done. This information is then sent back to the process layer. The DRL is realized as a rule-based system;

- the *reasoning layer* used for actual reasoning about decisions other than based on pre-defined set of behavior rules. When a need for more complex reasoning arises (e.g. prediction of what would happen if the driver made some maneuver), it has to be realized by the reasoning layer equipped with more sophisticated algorithms. The result is sent back to the DRL for further processing.

Several tools facilitating the system design have been implemented. In particular, we have created software shells for implementation of the two lower layers of the architecture as well as the languages for specification of software for those layers.

Similar layered architectures have recently been used for implementing autonomous real-time systems by Bonasso [1], Connell [2], Gat [3] or Steels [22]. Although there are differences both in the way of assigning various tasks to different layers and in the way the overall control of the system is executed, the general conclusion is that such layering is beneficial, if not necessary, in designing autonomous systems.

In the context of the DIU design we are discussing the following topics:

1. layering of the system;

2. real-time software engines: *Process Layer Executive* and *Rule Layer Executive*;

3. how this approach is used for implementing the system.

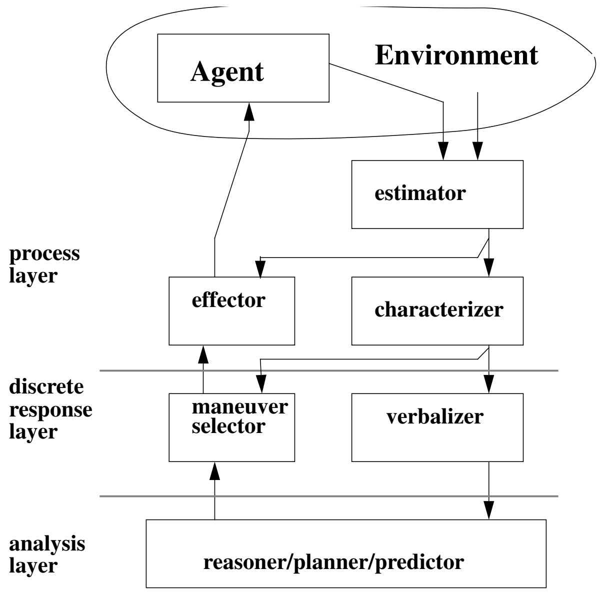


Figure 1: A three-layer architecture of an autonomous system.

## 2 A three-layered software architecture

The design of the driver information unit is based on the three-layered software architecture developed in our group [4, 20, 17, 16] for designing intelligent autonomous systems (see Figure 1). The three layers are separated on the basis of the type of computations they are intended to perform.

The first layer, called the *process layer*, is intended to be the place of implementing numerical, periodic tasks, such as preprocessing sensory data or controlling actuators. Our model of computations assumes this layer to be synchronous, although periods of execution of various tasks may differ. Tasks can communicate with each other either via the *state vector* or directly with each other. The state vector is also used to store preprocessed sensory information and computed control values for actuators. According to the AI naming conventions we will call the signals processed by the process layer *periodic fluents* representing the *piecewise continuous fluents* observed at the inputs and sent as the outputs of the system.

The middle layer is called the *discrete response layer* (or in some of our previous publications the *rule layer*) and is intended to perform tasks which are by their nature asynchronous, or in other words, data driven. An example of such task is the change of control mode (of the process layer) due to the change of situation on the road; e.g. car catching up the car in front, or car entering a motorway from a curving land road.

The computational model assumed for discrete response layer is that of discrete event systems (DES). There exist several equivalent DES formalisms: automata, transition systems, rule-based systems, etc. All of them distinguish the notions of *state* and *transition* as central, although the details vary from model to model. We have adopted the rule-based approach as the most convenient for the purpose of specification of a knowledge-based DES. This does not preclude usage of other approaches, what has been shown in [6, 7, 21].

The signals processed by this layer are called *piecewise constant fluents*.

The last layer is called the *analysis layer* because it is intended to handle *symbolic* reasoning tasks such as prediction, planning and replanning. As *limited prediction* does not require the full reasoning capabilities of the system, it might be distinguished as a sub-layer interfacing the discrete response layer. However, it still needs the ability to reason *about* time flow and its effects and therefore has to be included in the highest layer of the architecture. This layer is expected to produce controlling sequences for the rule layer. This layer would also host eventual learning mechanisms which might adapt the discrete response layer while maintaining its critical response requirements.

# 3 Real-time software tools

During our previous research done within the Prometheus programme we have thoroughly studied this architecture and its implications on software engineering issues [16]. One of the conclusions was that it facilitates prototyping of systems, especially because it allows development of generic software tools which can be used for implementation of each particular application system. Along this line we have developed software kernels, or *engines*, for development and implementation of the process layer and the discrete response layer. The set of tools includes:

- Process Layer eXecutive (PLX) [15]: a multi-threaded time-triggered real-time engine for implementation of process layer software. It has been implemented on a VME-bus system equipped with the pSOS$^+$ real-time operating system, and on a PC-compatible micro with VDX operating system. There also exists a simulator of the PLX which runs under Unix;

- Process Layer Configuration Language (PLCL) [10, 12] and its compiler: a language for specification of process layer module interconnections and interfaces to both sensors and actuators on one side, and to the rule layer software on the other side. The modules themselves are programmed in some conventional language, such as C or C++;

- Rule Layer eXecutive (RLX) [14]: an engine for implementation of rule-based discrete-event systems. It has been implemented as a LISP program on a Unix workstation and, recently, on a PC with VDX;

- Rule Layer Configuration Language [13]: a logic-based rule language for declarative description of discrete-event control imposed on process layer by the discrete response layer.

## 3.1 The Process Layer Executive

The process layer executive facilitates the implementation and maintenance of the hard real-time parts of the application which perform the transformations of periodic fluents in the characterizer and effector blocks.

During processing, fluents are represented by a *dual state vector*. The dual state vector is a global data structure in which every data item is stored in two versions, thus forming two subvectors; one which is read-only and one which is write-only. The values in the read-only vector represent either sensor readings or internal state. The write-only values represent actuator outputs or new internal state.

The use of a dual state vector for representing the fluents in the process layer allow the computation to distinguish unambiguously between the new and the old value of a parameter. If one just has one memory location for each parameter, it becomes necessary to choose the right sequencing of the transformations to obtain the correct results. By having a data representation that distinguishes explicitly between the old and the new value it becomes unimportant in which order the different state-vector components are calculated.

A process layer application defines a sequence of transformations that should be applied to the read-only vector in order to compute the new values of the write-only vector. Since sensor values are read periodically, the transformations have to be applied with the same periodicity. When the transformations have been applied, actuator outputs are flushed to devices and the values of the write-only vector are copied to the read-only vector, thus establishing the new state of the process layer. This operation is referred to as a *flip* of the state vector.

The *Process Layer Executive* (PLX) [15] is a software tool which maintains a dual state vector. This task includes performing the flips, and reading sensors and writing to actuators using user defined access functions. In addition, the PLX executes the transformations on the state vector in a periodical fashion. The PLX supports decomposition of the dual state vector into several subvectors, each of which has its own period, forcing transformations to be executed at different rates.

The *Process Layer Configuration Language* (PLCL) [12] is used to define the components of the dual state vector and to configure the user supplied library modules which implement the transformations on the fluents. Moreover, the PLCL description forms the basis for a worst case execution timing analysis of an application [11].

### 3.2 The Rule Layer Executive

The rule layer executive facilitates the rule-based discrete fluent to discrete fluent transformations of the selector block. The fluent is represented there by the time dependent values of a set of *slot-records*. As is the case in the process layer executive, it is important to be able to unambiguously distinguish between the new and the old value of a slot. The discrete nature of the rule layer fluent suggests that a slot should be updated only when its value changes (in contrast to the periodic updates in the process layer). To still be able to distinguish the old values from the new ones, each slot-record is realized as a data structure containing the new and old value as well as the time for the update.

A more operational view of the rule layer fluent may be described as follows. The fluent is uniquely determined by the initial slot values and a sequence of time-stamped *assignments*, where each assignment states how the value of a particular slot should be changed. Such assignments are the basic building blocks from which the *rules* are defined. A rule specifies dependencies between slots, typically of the form that if the value of a particular slot is changed in a certain way, then the value of another slot should also be changed as a result. Internally, each such assignment may trigger consequences, which are new assignments. This forward chaining process may continue in several steps.

We take an object-oriented view of the rule base in the sense that we associate a set of rules with each slot. This view facilitates the flexibility and maintainability needed for complex systems.

The *Rule Layer Executive* (RLX) [13] is a software tool which has two major tasks. Firstly, it defines and maintains the set of slots and rules which determines the behaviour of the rule layer. Secondly, it performs the forward chaining of rules. The forward chaining is typically initiated by an assignment "sent" from the process layer. The outcome of the forward chaining process may be an assignment "sent" to the process layer. In the process layer, those update requests may be used as parameters in control algorithms or for choice of control algorithm.

The *Rule Layer Configuration Language* (RLCL) [14] is essentially a syntactic variant of a simple logic defined in [6] and is used for declarative specification of behaviour of the rule layer.

We have employed the above architecture and tools for development of prototypical co-pilot systems in the context of the Prometheus project [8, 9]. In this case the "physical environment" consists of the own car, of other vehicles which surround the own car, and of the road network. The models of the environment at different levels of abstraction are based on vehicle dynamics, traffic rules, and normal and abnormal traffic behaviours. A number of driver support services were investigated. These included direct intervention in pre-defined hazardous situations e.g., obstacle avoidance, and issuing warnings when a proposed move by the driver, e.g., overtaking, is dangerous in the current dynamic environment.

# 4 A structured DIU design

The design of a DIU is presented here in a simplified setting. For a more concrete description the reader is asked to consult [5].

Let us assume that a car is equipped with several intelligent modules processing status data, visual information, communicating with the road environment and with other cars (or road users), gathering or producing road guidance information, etc. (An example of such vehicle can be a test car equipped with AICC-related modules, cf. [19].) The modules work in parallel, each of them performs numerical and synchronous computations, so all of them fulfill the requirements of a process layer application. What has to be added, in order to provide an intelligent interface between the driver and the underlying system, is an appropriately designed module which can possess knowledge about the driver, including his or her support needs, expectations and abilities, and about the dynamically changing traffic situation influencing the driver's workload and perceptual abilities. We observe that messages to the driver are generated asynchronously in the different modules and that the handling of output requests, therefore, conceptually belong to the discrete response layer.

The software development process for such a system looks as follows. The process layer applications (i.e. the intelligent application modules) are designed and implemented using any suitable set of software tools. We are advocating the usage of Process Layer Configuration Language and Process Layer Executive as the cyclic executive for this kind of tasks, but other (more conventional) solutions are also possible. As a result, the process layer consists of heterogeneous modules communicating asynchronously with the DIU residing in the discrete response layer. The programmer then specifies the rule layer application (i.e. the DIU) software in the Rule Layer Configuration Language (RLCL) and prototypes it on a platform of his choice: it can be either an MS-DOS microcomputer, or a Unix workstation. Then the programmer has the possibility of compiling it for one of the three target environments supported by the RLX (Unix, MS-DOS or VDX). In this way the whole process of software development is made easier and less error-prone. Moreover, the incrementality of the development process cannot be underestimated: to add another function the programmer has to add or modify only some RLCL rules rather than change the code of some particular (possibly low-level) function.

As an example, imagine that the vehicle is equipped with a set of modules providing information about the occupancy of the own and the meeting lanes. One of the process layer modules transforms this into a concise, discrete characterization of the current traffic situation. This characterization can have one of the following four values: "free", "soon-blocked", "blocked" and "soon-free". On the basis of this (and also some other) input the DIU is expected to inform the driver about the current situation. All that the designer has to do is to write a set of rules of the following type (this particular rule has been borrowed from [18]):

```
WHEN signal-me *= {off,left}
  IF
    position-me |= right-lane AND
    right-road |= soon-blocked AND
    left-road |= soon-blocked
  THEN
    road := soon-unsafe;
```

The message "soon-unsafe" is then translated into appropriate message (displayed on the screen or spoken out) to the driver (provided he is able to receive it, i.e. is not occupied by something more important).

Such set of rules can be compiled down into the machine code of the computer of programmer's choice: in case of prototyping it will probably be a workstation, in case of field test it will rather be a microcomputer with some suitable operating system, such as VDX.

We have used the described software to implement a simple prototype of a driver information unit which we plan to test in a test car in the nearest future [5]. The unit consists of more than 70 rules representing a subset of all the possible traffic scenarios. The purpose of the prototype is to estimate and maintain a model of the current maneuver of the own car, the current workload of the driver and to decide on this basis whether incoming messages should be presented to the driver, and how to do it.

The design adopted here has one more important advantage: it allows for almost any changes of the underlying process layer configuration. Changes in functionality of existing application modules and/or changes of the number of applications can be easily handled by merely changing an appropriate subset of the rules which describe the behaviour of the DIU. Moreover, the unit can be also modified to reflect the changes of functionality of output devices and addition of new functions.

# 5 Conclusions

The design process of any complex, heterogeneous, distributed real-time system (such as a car equipped with a set of intelligent modules moderated by a driver information unit) is a complex and challenging task. Another dimension of complexity is introduced by the fact that the hardware devices and the software, especially in test cars, are changed relatively often. Hence, the DIU has to function in a development environment and under the assumption of incrementality, both in terms of hardware (new modules are added) and software (new functions for the existing modules are implemented).

We have proposed a software engineering methodology based on experience and results from artificial intelligence. This methodology, and the associated design and software tools (languages and engines) are intended to make this task simpler and more manageable.

We have described a way of applying this methodology to the design of a Driver Information Unit. This solution seems very suitable since we have to deal with evolving systems whose structure is likely to change quite often. The main advantages of this approach are: easy adaptation to new and/or refined functions and portability. We aim at obtaining a predictable ( hard real-time) system, and we plan to concentrate more on this topic in the nearest future.

# References


[1] R. Peter Bonasso. Integrating reaction plans and layered competences through synchronous control. In *Proceedings of the Twelvth International Joint Conference on Artificial Intelligence, Sydney*, pages 1225–1231. Morgan Kaufman, 1991.

[2] Jonathan H. Connell. SSS: A hybrid architecture applied to robot navigation. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 2719–2724, Nice, France, May 1992.

[3] Erann Gat. Integrating reaction and planning in a heterogenous asynchronous architecture for mobile robot navigation. *SIGART Bulletin*, 2(4):70–74, 1991.

[4] Johan Hultman, Anders Nyberg, and Mikael Svensson. A software architecture for autonomous systems. In *Proc. 6th International Symposium on Unmanned Untethered Submersible Technology*, pages 279–292, 1989.

[5] IEEE Industrial Electronics Society. *Intelligent Vehicles'93 Symposium*, Tokyo, Japan, July 1993.

[6] Jacek Malec. Applied knowledge representation. *CC-AI: The Journal for the Integrated Study of Artificial Intelligence, Cognitive Science and Applied Epistemology*, 9(1):9–41, 1992.

[7] Jacek Malec. Complex behavior specification for autonomous systems. In *Proc. IEEE International Symposium on Intelligent Control'92*, pages 178–183, Glasgow, Scotland, August 1992.

[8] Jacek Malec, Reza Farahani, Magnus Morin, Simin Nadjm-Tehrani, Per Österling, and Erik Sandewall. An integrated software environment for development of driver support systems. In *Proc. of the Prometheus Pro-Art Workshop on Intelligent Co-Pilot*, pages 31–39, Grenoble, France, December 1991.

[9] Jacek Malec and Per Österling. Driver support system for traffic manoeuvers. In M. Bielli, G. Ambrosino, and M. Boero, editors, *Artificial Intelligence Applications to Traffic Engineering*, pages 283–318. VSP International Science Publishers, Zeist, The Netherlands, 1994.

[10] Magnus Morin. Process Layer Programmer's Guide, Version 2.0. Technical Report LAIC-IDA-90-TR34, Linköping University, 1990.

[11] Magnus Morin. Performance Aspects of the Process Layer Executive. Technical Report LAIC-IDA-91-TR13, Linköping University, 1991.

[12] Magnus Morin. PLCL – Process Layer Configuration Language. Technical Report LAIC-IDA-91-TR10, Linköping University, 1991.

[13] Magnus Morin. Building Rule Layer applications. The user's guide. Technical Report LAIC-IDA-92-TR18, Linköping University, 1992.

[14] Magnus Morin. RLCL — Rule Layer Configuration Language. Technical Report LAIC-IDA-92-TR17, Linköping University, 1992.

[15] Magnus Morin. Predictable cyclic computations in autonomous systems: A computational model and implementation. Licenciate thesis 352, Department of Computer and Information Sciences, Linköping University, 1993.

[16] Magnus Morin, Simin Nadjm-Tehrani, Per Österling, and Erik Sandewall. Real-time hierarchical control. *IEEE Software*, 9(5):51–57, September 1992.

[17] Magnus Morin and Erik Sandewall. A Software Architecture Supporting Integration of Sub-Systems. In Lars Asplund, editor, *National Swedish Symposium on Real-Time Systems*. Technical Report No. 30, Dept. of Computer Systems, Uppsala University, 1991. Also available as Technical Report LAIC-IDA-90-TR21, Dept. of Computer and Information Science, Linköping University.

[18] Simin Nadjm-Tehrani. Analysis of the overtaking scenario: Specification of an autonomous car and a driver support system. Technical Report LAIC-IDA-91-TR1, LITH-IDA-R-91-07, IDA, 1991.

[19] Ulf Palmqvist. Intelligent cruise control and roadside information. *IEEE Micro*, pages 20–28, February 1993.

[20] Erik Sandewall. System Block Specifications and their PROMETHEUS applications. In *Prometheus: Proceedings of the Fourth Workshop*, 1990. Available also as LAIC-IDA-90-TR22.

[21] Hua Shu and Jacek Malec. From Process Transition Networks to Behavior Automata. In *Proceedings of the 1991 IEEE International Symposium on Intelligent Control, Arlington, VA*, pages 257–262. IEEE Control Systems Society, 1991.

[22] Luc Steels. Building agents out of autonomous behavior systems. In Luc Steels and Rodney Brooks, editors, *The "artificial life" route to "artificial intelligence". Building situated embodied agents.* Lawrence Erlbaum Ass., 1993.