A Layered Software Architecture for Design and Analysis of Embedded Systems

Jacek Malec, Magnus Morin, Simin Nadjm-Tehrani*
Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden
jam,mmo,snt@ida.liu.se

Abstract

For several years we have been working on an architecture for modelling and implementing autonomous agents [9]. This work has been recently extended in two directions: evaluating the benefits of the architecture in a real-world setting in terms of ease of design, rapid prototyping, dynamic resource management, and studying methods for formal analysis of the reactive subsystem based on the composition with models of its physical environment. This paper gives an introduction to the architecture and a flavour of the above-named extentions.

1 Introduction

In this paper we describe a framework for design, implementation and analysis of hierarchical embedded systems. Organizing different types of computation in several layers is a well-known and common approach for development of intelligent autonomous systems. The work we report on here is based on such a layered software architecture developed in our group [5, 9] since 1986. The paper reports on some real-time applications developed based on this architecture and elaborates on the use of the architecture as a tool for decomposition of a system into modular mathematical descriptions for the purpose of formal verification of its properties.

The layered architecture hosts both synchronous and asynchronous computations and provides:

- an environment for the development of resource bounded planning and control programs;
- means for calculation of worst case computation time for individual application components;
- a guide to natural decompositions in formal models of concurrent reactive programs and their interface to the physical environment.

This architecture and the tools associated with its various layers have facilitated the development of prototypes in an industrial setting [?]. The experiments have in turn provided insight into the types of properties of interest for analysis purposes. The paper describes two different applications of the architecture. The driver information unit case study (Section 4) is used to illustrate the design principles, whereas the anti-collision system study (Section 5) illustrates how formal models of systems subject to analysis can benefit from the architecture introduced here. The motivation for the latter work is to introduce a notion of 'correctness' in the context of embedded systems which perform a combination of symbolic and numeric computations.

In order to define an appropriate notion of correctness there should of course be a notion of intended behaviour. It is intuitively clear that the real-world dynamics place an obvious demand on the performance of an embedded system. Any notion of correctness for such autonomous systems should therefore draw on related notions from the fields of control theory and real-time systems verification. The paper gives a brief account of how models for reactive software in a real-time application can be composed with models of its environment using some of the system blocks as an interface. Our experience shows that the composed model obtained in this way can then form the basis for application of formal verification techniques.

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

2 The Layered Software Architecture

Our approach to hierarchical embedded systems builds on the three-layered software architecture developed in our lab [5, 9]. The architecture groups similar types of computations into *system blocks* (shown in Figure 1). We identify the following blocks:

- *Estimator:* samples continuous signals in the environment.
- *Adapter:* manages output hardware.
- *Effector:* implements basic control algorithms.
- *Characterizer:* monitors sampled data to detect significant events.
- *Selector:* controls the operation of the effector by responding to events.
- *Verbalizer:* monitors charcterizer output to detect situations that require reasoning.
- *Reasoner:* hosts planning and scheduling which control the selector.

The three layers are distinguished on basis of the type of data they process.

The first layer, called the *process layer*, is intended to host implementations of numerical, periodic tasks, such as identification or control. Data handled by this layer is contained in input and output vectors. Computations have the form of mappings from input vectors to output vectors and are performed periodically in synchronization with the sample rates of sensors.

The middle layer is called the *discrete response layer*, and perform tasks which are by nature asynchronous. For instance, it computes the response to asynchronous events that are recognized in the process layer. An example of such a task is the change of control mode (of the process layer) due to the change of mode in the environment. The computational model assumed for the 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 used the rule-based approach for the purpose of specifying knowledge-based system prototypes. However, this does not preclude usage of other approaches [?, ?, ?].

The last layer is called the *analysis layer* because it is intended to handle *symbolic* reasoning tasks such as prediction, planning and scheduling, which require reference to physical time. The output of this layer can be either control events that guide the selector in the discrete response layer or parameter settings that are passed through the discrete response layer and directly affect the operation of the effector in the process layer.

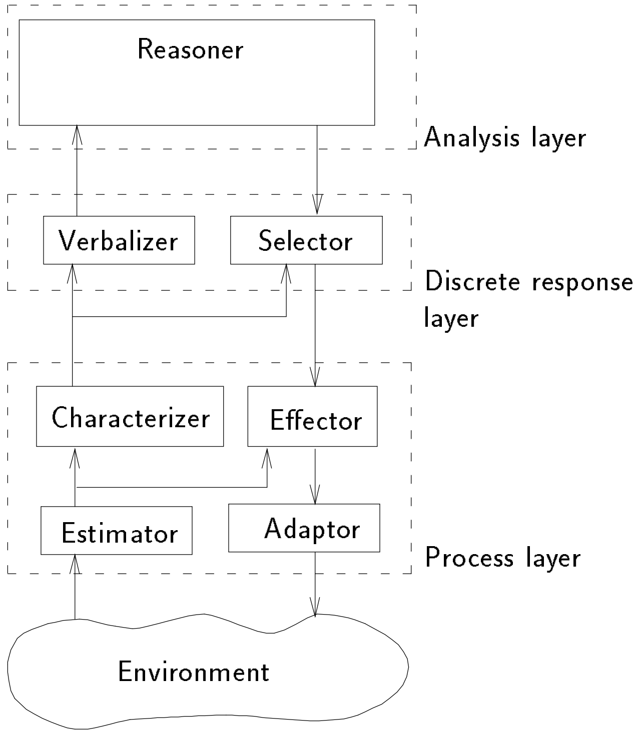


Figure 1: A three-layer software architecture for embedded systems.

3 Real-Time Software Tools

During our previous research we have thoroughly studied this architecture and its implications on software engineering issues [9]. 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) [7]: 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 There also exists a simulator of the PLX which runs under Unix;

- Process Layer Configuration Language (PLCL) [6] 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 discrete response layer software on the other side. The modules themselves are programmed in a subset of some conventional language, such as C or C++;
- Rule Layer eXecutive (RLX) [8]: 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 both VDX and MS-DOS;
- Rule Layer Language (RLL) [8]: 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 PLX supports the implementation and maintenance of the hard real-time parts of the application which perform the transformations of periodic data in the characterizer and effector blocks.

During processing, all data is stored in a *dual state vector*, which is a global data structure consisting of an input and an output vector. The values in the input vector represent either sensor readings or internal state, whereas the values in the output vector represent actuator outputs or new internal state.

A process layer application defines a sequence of transformations that should be applied to the input vector in order to compute a new output 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 new internal state is installed in the input vector, thus establishing the new state of the process layer.

The PLX engine supervises a process layer application, which includes managing the internal state, reading sensors and writing to actuators using user defined access functions, and supervise the execution of the periodical transformations of the dual state vector. The PLX supports decomposition of the vector into several subvectors, each of which has its own period, causing transformations to be executed at different rates.

The PLX engine and the PLCL compiler together form the basis for a worst case execution timing analysis of a process layer application [7].

3.2 The Rule Layer Executive

The rule layer executive supports the implementation of rule-based event processing in the verbalizer and selector blocks. In the RLX the state of the world is represented by the time dependent values of a set of symbolic state variables called *slots*. A slot is updated only when its value changes, due to an external event or as a result of a change of another slot's value. Rules specify dependencies between slots, and typically have the following form: *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 change may trigger additional rules, which lead to more updates. 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 RLX tool has two major tasks. Firstly, it maintains the set of slots and rules which determines the behaviour of the discrete response layer. Secondly, it performs the forward chaining of rules. The forward chaining is typically initiated by an event recognized by the characterizer. The result of the forward chaining process may be the change of control algorithm in the effector or direct output to a device interfaced by the adapter.

The RLL is essentially a syntactic variant of a simple temporal logic defined in [?] and is used for declarative specification of behaviour of the discrete response layer.

4 A Case Study: Design

When extending a complex embedded system with new functionality the design alternatives are limited by the fact that large parts of the system have already been designed and implemented and cannot be changed without great effort and considerable costs. An architecture for such systems therefore has to accommodate both the already existing components, even though their internal details may not be known, and the ones to be added. In the following we describe how our development of a driver information unit for the AICC experimental vehicle has benefitted from the use of the software architecture introduced earlier in this paper.

4.1 The AICC system

Today's cruise control systems for cars are simple, in the sense that they control the car according to the driver's pre-set speed only, without considering other relevant factors such as the speed limit of the road, the distance and the relative speed to the car in front. The purpose of *autonomous intelligent cruise control* (AICC) is to improve this situation.

In the AICC experimental car [12] there are several sensors that provide information about the status of the car and about its environment. Most sensors are sampled periodically and their output data is broadcast on the on-board network where it is available for use by different applications. The basic level of information processing in the car is performed by local feed-back controllers for engine, brakes, steering, etc. They capture periodical sensor data from the on-board data bus, compute control signals and send them through the network to the actuators. Many local controllers are black boxes supplied by the manufacturers of the physical devices they control. In the AICC experimental vehicle a supervisory controller computes parameters for the engine and brake controllers based not only on the driver's pre-set speed, but also on the speed-limit of the road and the distance and speed of the car in front. This is possible due to the use of a sophisticated on-board radar system and car-roadside communication.

4.2 Driver Information Unit

Of course the AICC system has to inform the driver about its decisions, for instance the reason why the AICC system drives with a lower speed than the one set by the driver. In addition, other technical systems in the car, such as route guidance and road-tyre friction estimation need to communicate with the driver. The situation is further complicated by the fact that the amount of information generated varies dynamically, the amount of information that can be presented to the driver is limited, as is the amount of information the driver can conceive. This calls for a systematic solution of the driver information problem which takes all these factors into account.

The *Driver Information Unit* (DIU) [?] acts as a mediator between different applications by selecting the most urgent pending information requests for presentation to the driver. It monitors the state of the system by listening to the on-board data bus. When the DIU detects important changes in the data that define system events, it processes the event and updates its internal model of the car, the traffic environment and the driver, and determines if the driver should be notified. If that is the case the DIU examines what resources, in terms of time and output channels, are required to present the information. It determines whether the information should be presented to the driver immediately, possibly preempting another message, be queued for later presentation, or be discarded.

In the design of the DIU we used the software architecture as a means of separating different types of computations to gain an understanding of both the existing AICC system and the requirements imposed by the DIU. The first part of this undertaking consisted of casting the existing components of the AICC system in terms of system blocks as to establish the interface to the DIU. It turns out that essentially all processing in the existing AICC car is readily modelled by the estimator, effector and adapter blocks. Sensors provide the interface to the actual car and belong to the estimator block of the software architecture. The computations performed by the feed-back controllers and the speed supervisor operate on periodic input and generate periodic output and therefore belong to the effector block. Output filtering and low-level hardware control is handled by the adapter block.

The DIU is naturally modelled as an event-processing system, so it actually extends the system by adding a selector and its corresponding characterizer. Also, it adds functionality to the adapter for carrying out low-level control of the physical devices used to present information to the driver.

We have devised two DIUs based on different principles. For the first one, we assumed that the driver information task would be carried out based on the change of state variables only, without reference to physical time. The purpose of this, it may seem overly optimistic, assumption was to rapidly arrive at a first implementation of the system which could be tested together with the rest of the car. This DIU involves the selector and characterizer blocks only and has been implemented using the RLX tool.

Of course a realistic driver information unit has to take time into account to provide appropriate driver information. Since the selector deals only with state transitions, and lacks the ability to reason about time, it cannot on its own support a fully-fledged DIU. In the second DIU design we extend the system to also include a scheduler which dynamically determines if and how requests for driver information could be handled, taking their resource requirements into acoount. Conceptually, the scheduler belongs to the analysis layer since it extends the discrete response processing in the selector block with reasoning about physical time. A verbalizer is also required to monitor inputs from the characterizer and notify the scheduler whenever a state which requires output is reached. The selector is being modified to receive inputs both from the characterizer and from the scheduler. It turns out that both the verbalizer and the modified selector can be implemented in the rule-based framework supported by the RLX.

4.3 Discussion

Literally dozens of layered architectures have been proposed recently for implementation of intelligent real-time systems, either embedded or autonomous (see e.g. [1, 2, 3, 4, 13] for just some of them). 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 achieving desired level of intelligence of the designed system. We do not claim that the architecture presented here is beneficial per se, rather our claim is that its advantage for the design of an embedded system is twofold:

1. By explicit separation of tasks requiring different conceptual and computational frameworks;

2. By providing a potential designer with a set of tools simplifying and systematizing the design process itself.

We would especially like to stress the latter property.

Our work on a DIU for the AICC car has benefitted in several ways from the use of the software architecture presented earlier. First of all, it provided a clear conceptual structure for the design and implementation. By providing a subdivision of the overall system design into blocks based on the type of computations performed it facilitated our understanding of the existing AICC system and made it easy to establish the interface the DIU and the rest of the AICC vehicle.

In our first DIU prototype we assumed a simple model for the interaction with the physical world. In spite of the fact that it does not support reasoning, it still turned out to be useful as a basis for rapid prototyping of the DIU using the RLX tool. The highly incremental task of developing user interfaces was greatly supported by the rule language used in the RLX.

As new sensors were added to the AICC car the DIU was easily adapted. The characterizer was changed to recognize the new types of information available and the rules in the selector was updated to include this information in the decision making process. It is our experience that the architecture facilitates system modifications by providing a robust framework and modularization.

5 A Case Study: Analysis

A major question of interest for embedded systems is whether the reactions of the system are 'adequate' with respect to the dynamics of the surrounding environment. A large and growing volume of work for specification and verification of time-dependent properties of computer systems [?] already exists. A recent trend within software verification is to stretch the existing methods for discrete systems to formalisms which allow explicit representation of continuously changing variables [?]. The models discussed in the field are, however, not derived from various components in an application in a systematic manner. In control engineering, on the other hand, an opposite trend is observable: for some years the attempt to abstract from traditional models for continuous components has been pursued within the subarea of discrete event systems. There it is normal to derive models of a 'whole' from the mathematical descriptions of its 'parts'.

5.1 Our Analysis Framework

In our work we emphasize the need for bringing the interface between the continuous and the discrete worlds into focus. Using an appropriate interface, we treat the surrounding environment *on a par* with the reactive system which is subject to analysis. By reactive system it is meant the blocks residing in the lower layers of the architecture as depicted in Figure 2.

This approach can be compared with the traditional software verification approach where the subject of verification is the computer system. The model of environment, to the extent that it is present, is an ad-hoc abstraction of aspects of the physical world.

For the reactive software, we use variants of transitions systems for which some formal verification techniques are known. Then the formal model of a selector in terms of a transition system is in direct correspon-

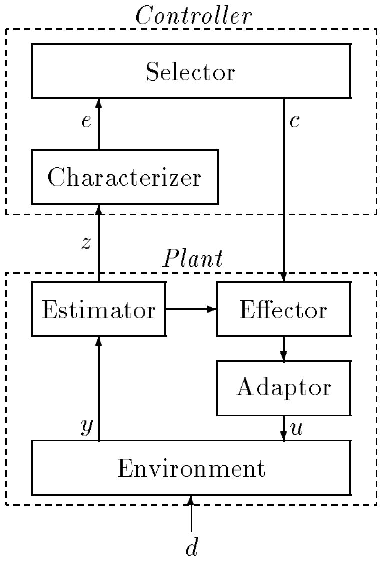


Figure 2: Our analysis framework

dance with the rule-based language we use for developing the discrete response layer.

For the environment, we advocate the 'reuse' of physical models obtained using well-known modelling techniques. This is beneficial due to the following reasons [10]. First, these models already exist and form a basis for the control algorithms implemented in the effector block of our architecture, and therefore need not be developed from scratch. Second, the result of formal verification of a model of a system is only as reliable as the model itself. Since formal verification of timed concurrent processes is a tedious task, it is utterly important that the models used in the verification process adequately reflect the reality of the system in question. In our modular variants of transition systems the physical model of the environment is systematically composed with a discrete model of the discrete controller. An instance of such *Hybrid Transition Systems (HTS)* is provided in the following example [10, 11] .

5.2 Anti-collision System

Our work on formal analysis of the reactive subsystem has so far concentrated on simplified versions of real-world applications which nevertheless remain non-trivial. Consider the scenario in Figure 3 consisting of the two cars A and B each with its own driver, and no restrictions on their driving behaviour. A support function for obstacle avoidance is implemented in car A which drives behind car B.

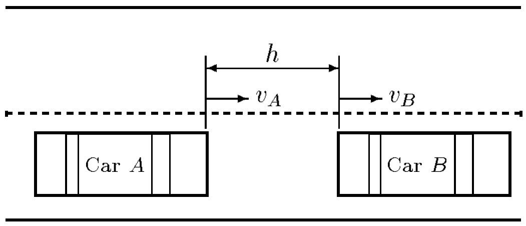


Figure 3: The two car scenario.

We assume that there are only two modes of operation determined by the selector within the reactive support system. We further assume that each mode in the set $\mathcal{C} = \{c_0, c_1\}$ of control modes corresponds to algorithms (control actions) informally described as follows:

c_0 : no intervention by the support system, i.e. driver in car A in charge,

c_1 : maximum braking exercised by the support system, i.e. acceleration (or deceleration) by the driver in car A disconnected.

The goal of verification is to show that the reactive software in interaction with its environment satisfies predefined requirements. In this case, an obvious requirement would be to show the safety property 'no crash'. That is, whatever the behaviours of the respective drivers are, the two cars will never crash under sequences of control modes selected from the above repertoire. Note that the requirement is stated as a constraint on a variable from the (physical) environment, but its correctness is to be proved in the composed system including the controller. Using the physical variables involved in Figure 3 within the framework depicted in Figure 2, we complete the physical model as follows.

Disturbances: We introduce two disturbance signals, namely $d = \langle d_A, d_B \rangle$ where d_A (d_B) $[m/s^2]$ is the driver commanded acceleration/deceleration of car A (B).

Environment input: There is only one input available for manipulation, namely $u = \langle u_A \rangle$ where $u_A[m/s^2]$ is the actual acceleration/deceleration of car A.

Effector: The effector acts as a simple switch function:

$$u_A = \begin{cases} d_A, & \text{if } c = c_0 \\ -d_{max}, & \text{if } c = c_1 \end{cases}$$

for some positive real constant deceleration d_{max} $[m/s^2]$.

Environment model: We introduce the following three state variables: $x = \langle v_A, v_B, h \rangle$, where v_A (v_B) is the velocity for car A (B) and h is a measure of the relative distance between the two cars (see Figure 3). Thus we get the following set of linear differential equations:

$$\begin{aligned} \dot{v}_A &= u_A \\ \dot{v}_B &= d_B \\ \dot{h} &= v_B - v_A \end{aligned}$$

Note that although we do not have any *a-priori* knowledge about how the actual disturbance signal d evolves in time, the equations above still show the influence of disturbances on the state variables.

Environment output: The monitored aspects of the environment, i.e. y, is solely defined by the available sensors. Clearly it has to include at least the state variable h needed to express the property 'no crash'.

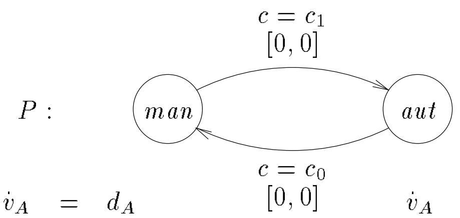
Finally, we introduce some reasonable constraints as to how the physical variables may vary: $0 \leq v_A \leq v_{max}, 0 \leq v_B \leq v_{max}, -d_{max} \leq d_A \leq d_{max}$ and $-d_{max} \leq d_B \leq d_{max}$ for some positive real constants $d_{max}[m/s^2]$ and $v_{max}[m/s]$.

Characterizer:

The interface between the plant and the selector consists of a characterizer based on a real-valued function ϑ. Assuming that all the state variables (h, v_A and v_B) are measured, the characterizer compares the value of $\vartheta(v_A, v_B)$ with the distance h in order to determine the discrete events.

The plant model

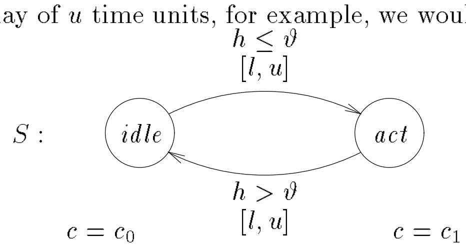
The following transition graph P represents a model of the plant with state variables $\{v_A, v_B, h\}$ and input variables $\{c, d_A, d_B\}$. Note, however, that this model places no restrictions on the characteristics of the input variables as a function of time. Each mode in the plant is governed by the associated set of differential equations, and a discrete change of mode takes place based on the charachterizer events with zero delay.



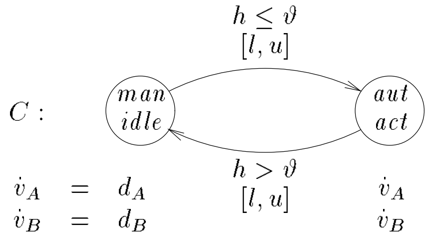
$$\begin{aligned} \dot{v}_A &= d_A \\ \dot{v}_B &= d_B \\ \dot{h} &= v_B - v_A \end{aligned} \qquad \begin{aligned} \dot{v}_A &= -d_{max} \\ \dot{v}_B &= d_B \\ \dot{h} &= v_B - v_A \end{aligned}$$

Timed model of the selector

Adding time to the selector model would typically be straightforward; we simply add lower and upper time bounds to the transitions in the discrete transition system corresponding to the rules residing in the rule layer. Methods for ensuring bounded response time in the layered architecture would then facilitate arriving at lower and upper bound on transitions [9]. Assuming a minimal delay of l time units and a maximal delay of u time units, for example, we would get:



Using the composition binary operator defined for HTS the model of the complete closed loop system can then be derived as follows:



$$\begin{aligned} \dot{v}_A &= d_A \\ \dot{v}_B &= d_B \\ \dot{h} &= v_B - v_A \\ c &= c_0 \end{aligned} \qquad \begin{aligned} \dot{v}_A &= -d_{max} \\ \dot{v}_B &= d_B \\ \dot{h} &= v_B - v_A \\ c &= c_1 \end{aligned}$$

5.3 Discussion

It can be seen that the above model is a compact and formal description of the environment, the charcterizer, the selector and the effector in this application. This model can be subjected to formal verification techniques available for hybrid systems [10]. In general, such verification draws on elements of analysis from control theory and logical reasoning involving all discrete transitions. In this example it can be shown that if ϑ is chosen to be greater than

$$\vartheta_{min} = \max\{\frac{1}{2}(v_A^2 - v_B^2)\ d_{max}^{-1}, 0\}$$

then an analysis of the solution to the differential equations shows that provided the initial distance between the cars is $> \vartheta_{min}$ and full braking is exercised for $h \leq \vartheta_{min}$ then the distance between the two cars is always > 0, i.e. the two cars can never collide. Hence, the required safety property can be formally verified. The model can then form the basis of a software implementation of such a device.

6 Conclusions

A main goal of our work with the layered software architecture has been to bring design and analysis into the same framework. This would greatly facilitate the development of correct software for hierarchical embedded systems. In this paper we have presented two applications and demonstrated how the architecture, and the software tools and analysis methods provided, have been useful in this respect.

References

[1] J. S. Albus, H. G. McCain, and R. Lumia. NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM). Technical report, National Institute of Standards and Technology, 1989.

[2] R. Peter Bonasso. Integrating reaction plans and layered competences through synchronous control. pages 1225–1231.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] Magnus Morin. RL – RL: An embedded rule-based system. Technical Report LAIC-IDA-94-TR2, Linköping University, 1994.

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

[10] Simin Nadjm-Tehrani. *Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification.* PhD thesis, Department of Computer and Information Science, University of Linköping, 1994.

[11] Simin Nadjm-Tehrani and Jan-Erik Strömberg. From Physical Models to Compositional Modelling of Hybrid Systems. In *Proc. of the 3rd. International Conference on Formal Techniques in Real-time and Fault-tolerant Systems, LNCS 863*, pages 583–604. Springer Verlag, 1994.

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

[13] Lee Spector and James Hendler. Knowledge strata: Reactive planning with a multi-level architecture. Technical Report CS-TR-2564, Department of Computer Science, University of Maryland, College Park, MD, November 1990.