Real-Time Hierarchical Control

MAGNUS MORIN, SIMIN NADJM-TEHRANI, PER ÖSTERLING, and ERIK SANDEWALL, Linköping University

◆ *This layered framework supports the design and implementation of control systems with both continuous and discrete components. It has been used to develop a driver-safety system for automobiles.*

The last decade has seen the development of highly complex, often distributed, systems that can monitor, support, or control dynamic systems. The spectrum of problems associated with these systems — which range from industrial process-control and medical-support systems to intelligent robots and autonomous vehicles — has resulted in specialized research, the results of which are often not used in other areas. Some research is concentrating on formal methods for specifying and verifying real-time aspects, for example, but it has had little effect on architectural and implementation issues.

Work in the artificial-intelligence community on autonomous systems is facing similar problems. Researchers concerned with finding efficient solutions to implementation problems or symbolic representation issues have paid little attention to guaranteeing system-response time, which is important in applications like expert control systems. Attempts have been made to incorporate symbolic manipulation in traditional control systems, which allows reasoning at a higher level of abstraction and the expression of qualitative control knowledge. However, attempts to couple expert systems ad hoc have not worked well because there is no mechanism for handling reasoning about time.[1]

The goal of the framework we propose is to support the design and implementation of control systems that combine continuous and discrete components — which eliminates the need to combine expert systems in this haphazard fashion. Systems designed with this framework can provide solutions that combine both numeric and symbolic computation.

0740-7459/92/0900/0051/$03.00 © IEEE

COMBINING SYMBOLIC AND NUMERIC

The framework we propose is suitable for developing systems with both soft and hard real-time requirements. In a hard real-time system, designers must consider the worst-case behavior and guarantee that the system meets its deadline. A soft real-time system uses "best effort" techniques to try to meet as many deadlines as possible. It also uses methods to handle violated deadlines.

The basic idea behind our framework is to capture an application's time demands and transform them to time constraints on layers. The layers group transformations from flows of sensor data to flows of actuator commands on different levels of abstraction. At the lowest layer are numerical models and algorithms, which are used to perform basic feedback control and build the symbolic models used in higher layers. The middle layer uses more abstract symbolic models to decide which algorithms are to be executed in the lowest layer. It also supplies the information needed to build higher level models in the top layer. The top layer reasons about time using symbolic models of the environment and influences the behavior of the middle layer over time.

Each layer is served by one implementation tool, which maintains application data and manages the transformations performed on that data. We have made these tools simple, with well-known characteristics, which lets us verify overall system properties using the properties of the individual tools. We developed these tools because we believe the way to validate hard real-time constraints in complex systems is to build systems with simple and well-understood components. We have tested these implementation tools and methods within the Prometheus project, described in the box on p. 55. The main benefit of our approach is that it considers time aspects in a system that combines numeric and symbolic processing. We can specify and validate the timing requirements for both the implemented tools and the application system. In a typical control system, data from sensors provides snapshots of the environment, which is essentially instantaneous information. However, to detect changes and discontinuities and monitor long-term trends, the system must also take a *series* of snapshots into account. It does this by analyzing numeric data, which in turn reveals discontinuities in the environment. This look at the big picture is vital if the system is to reason about the real world.

Our framework captures an application's time demands and adds them to time constraints on layers.

In our layered framework, synchronous processes monitor the environment periodically and notify the reasoning processes when a discontinuity occurs. Our approach differs from the layered autonomous agents developed by AI researchers such as Rodney Brooks,[2] which rely on emergent functionality. In Brooks' approach, autonomous agents are built bottom up from asynchronous layers with no use of symbolic representation. Although the separation of functionalities has led to efficient implementations, the design has not been proven to scale up to highly complex systems. Another researcher, Erann Gat, has suggested a layered architecture that hosts both symbolic and numeric computations.[3] However, neither Gat nor Brooks attempts to guarantee time constraints.

RECONCILING DATA VIEWS

A hierarchical control system must combine techniques from both control and software engineering to deal with complex, dynamic environments. Control engineering provides numeric methods and tools to control feedback. Software engineering offers programming languages and techniques for symbolic manipulation, which can be used to extend the applicability of the control-engineering methods.

Combining these two areas is not straightforward, however, because each perceives time-dependent data differently.

Control engineering naturally uses the concept of a signal, typically a vector of real numbers, which is a function of time. Software engineering, on the other hand, favors complex data structures, which in themselves have no inherent time relation but are modified using updates or assignment statements.

To have a coherent methodology that satisfies both views, we use the signal concept as the basis for software design, but generalize it to allow an arbitrary data structure as a function of time. Following AI terminology, we call the general signal a *fluent*.

Fluents define the flow of data within and between software subsystems. Each subsystem performs transformations on fluents, both theoretically and in the actual implementation. There are several types of fluents:

- *Piecewise continuous.* These fluents have a finite number of discontinuities in every bounded time interval. Piecewise continuous fluents describe quantitative aspects in dynamic environments, for example distances between a specific car and surrounding vehicles, their velocities, and their accelerations.
- *Piecewise constant.* These fluents are the same as piecewise continuous fluents except that there are a finite number of value changes in every bounded time interval. An example is expressing the qualitative distance between the driver's car and the car ahead as far, close, very close, or not existing.
- *Regular.* These fluents are piecewise constant over intervals of equal length. They are the result of approximating piecewise continuous fluents by sampling at fixed (short) intervals. Regular fluents provide a representation of piecewise continuous fluents that is suitable for a computer. An example is the sequence of distance measurements that approximate the continuously changing distance between two cars.

Each fluent requires a different type of computation. Computations that involve regular fluents typically consist of (rapid) cyclic transformations. These transformations are performed periodically, whether or not the values have changed. Computations on slow-changing piecewise constant fluents, on the other hand, should occur only when the fluent value has changed. That is, they should occur asynchronously when triggered by an event.

ARCHITECTURE

Our framework for developing hierarchical control systems consists of layers that group transformation types, the tools to help implement the layers and validate timing properties, and the mechanisms for communicating among layers.

Transformation layers. Figure 1 shows how transformation types are grouped in blocks and how data passes between them. The blocks are (from the bottom up)

- ♦ *Estimator.* Transforms piecewise continuous fluents representing sensor values to corresponding regular fluents through sampling.
- ♦ *Adapter.* Converts effector output to a form suitable for actuators.
- ♦ *Effector.* Implements the basic control algorithms.
- ♦ *Characterizer.* Classifies the current situation and detects significant events.
- ♦ *Selector.* Determines the control algorithms to be used in the effector.
- ♦ *Verbalizer.* Classifies events to obtain higher level descriptions.
- ♦ *Reasoner.* Produces control sequences (plans) for the selector.

To implement this structure, we grouped blocks with similar computational characteristics into three layers. This three-layer architecture is an extension of the ideas adopted in the Prometheus project.[4]

- ♦ The *process* layer consists of the estimator, the effector, the adapter, and the characterizer. It supports transformations on regular fluents and therefore operates cyclically. Piecewise continuous fluents — because they exist mainly in the system's physical environment — are approximated through regular fluents. The estimator handles the conversion from piecewise continuous fluents to regular fluents; the adapter handles the conversion from regular fluents to piecewise continuous fluents.
- ♦ The *rule* layer consists of the selector and the verbalizer and supports transformations on piecewise constant fluents. It aids in the implementation of complex decision algorithms through a rule-based paradigm.
- ♦ The *analysis* layer consists of the reasoner. It also supports transformations on piecewise constant fluents, but uses temporal reasoning (such as prediction or planning) and a temporal database.

The layers, in turn, group transformation types according to levels of abstraction.

To understand the rationale behind these divisions, consider the layers as they might apply to a driver-support system. Using sensory information, a driver-support system reasons about the driver's current activities in a dynamic environment influenced by an arbitrary number of other drivers. The types of support might include

- ♦ *Low-level reactions to changes in numeric parameters.* An example is regulating the accelerator to keep the car at a given distance from the car in front.
- ♦ *Warnings.* An example is to issue warning signals to the driver when it is unsafe to change lanes during passing.
- ♦ *Suggestions.* An example is suggesting to the driver to pass only within a safe distance from a bend or to complete the pass within x amount of time.

To make these decisions, the support system must build and maintain a series of environmental models at increasing levels of abstraction, from numeric to symbolic.

For low-level reactions, the system needs numeric models, which require computations involving regular fluents performed in the process layer.

To issue warnings, the system needs a symbolic environmental model. The model begins construction in the process layer, at the lower levels of abstraction. The resulting symbolic representations of recognized events are reported to the rule layer, which uses them to build its environmental model. It then uses that model to perform reasoning, which leads to a choice of internal reaction. Finally, it communicates the chosen internal reaction to the process layer, which transforms it to a control algorithm.

Although the process layer could theoretically determine the selected algorithm, this decision, because of its asynchronous nature, is more suitably handled by the rule layer.

To offer suggestions, the system must understand a given scenario on a higher level — involving the analysis layer — which requires that it have a deliberative nature. The analysis layer must let (complex) maneuvers be represented as primitive concepts, and should have an awareness about time.

For the system to recognize an activity

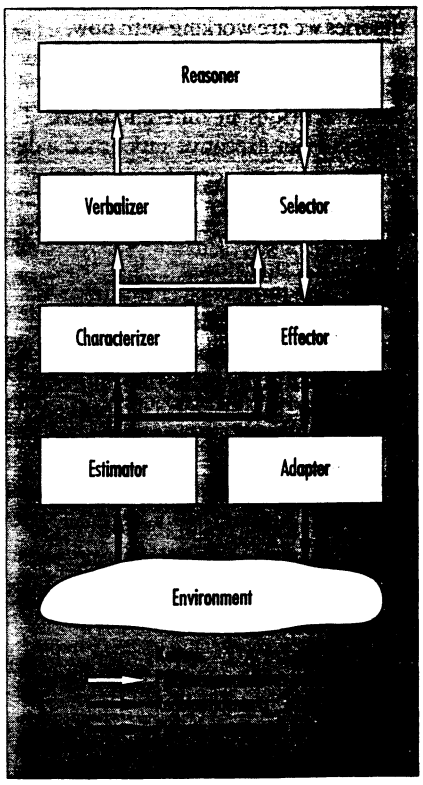

Figure 1. The fluent transformation blocks and fluent type received by each block.

that takes place over time, it must have access to the history of some fluents' values. Unlike the rule layer, which decides an internal reaction to an event without concern for how that event relates to other events (and therefore does not need an extensive value history), the analysis layer uses event sequences as its primitive concepts. Occasionally, event sequences may cause the system to change the set of rules that the rule layer consults. The system may recognize the need to switch between rule sets, or the change may occur because the system has learned something and recognizes an opportunity to optimize the rules.

The modularity and simplicity of our tools makes it easy to verify that computations will meet their deadlines.

Implementation tools. At present, we have tools that implement the process and rule layers. We plan to develop a tool for the analysis layer that is based on the formal theories we are working with now.

Time constraints and internal fluent representations vary according to the different activities in different layers. The process-layer executive supports the definition and execution of application software in the process layer; the rule-layer executive does the same for the rule layer.

PLX. The process-layer executive makes it easier to implement and maintain parts of the application that transform regular fluents in the characterizer and effector blocks. It realizes regular fluents in a collection of frames organized in a dual-state vector. In one state, the vector contains the current values of fluents; in the other, it stores the next value of the fluents during the same period. The first state is read-only; the second, write-only. The values in the read-only vector represent either sensor readings or internal state. The write-only values represent actuator outputs or a new internal state.

In a process-layer application, processes apply transformations to the read-only vector to compute the new values of the write-only vector. Because the state vector has this dual nature, processes are assured of accessing a well-defined state throughout the period. Moreover, the dual-state vector is the only way for interprocess communication, which means that the order in which communicating processes execute is irrelevant.

At the end of each period, PLX flushes output values to the adapter, and copies the values of the write-only vector to the read-only vector, thus establishing the new state. This operation is referred to as a *flip* of the state vector.

One of PLX's main tasks is to perform flips (maintain the dual-state vector), read estimator outputs, and write to the adapter. Another task is to supervise process execution and detect if time constraints are violated during runtime.

PLX also supports the dual-state vector's decomposition into several subvectors, each of which is flipped at the end of its corresponding period, forcing transformations to be executed at different rates. Each cycle's period must be a multiple of all shorter periods in the process layer. In this way, we guarantee that the (complete) state vector will be in a well-defined state periodically.

PLX is implemented in C and runs under Unix and the real-time operating system pSOS$^+$. It uses preemptive, priority-driven scheduling that is rate monotonic. Tasks with shorter periods have a higher priority. PLCL, the process-layer-configuration language, defines the components of the dual-state vector and aids in configuring user-supplied library modules to implement fluent transformations.

RLX. The rule-layer executive has two major tasks. It defines and maintains the slots — set of symbolic state variables — and rules that determine the behavior of the rule layer, and it performs the forward chaining of rules.

In this layer, fluents are represented as the time-dependent values of a set of slots. The state is treated as a fluent, and the fluent value at any time is the collection of all slot values. The state is uniquely determined by the initial slot values and a sequence of time-stamped assignments — each assignment prescribing how the value of a particular slot should be changed. Such assignments are the building blocks for defining rules. A rule specifies dependencies between slots: if the value of a particular slot is changed in a certain way, then the value of another slot should be changed as well.

Internally, each value assignment may trigger consequences, which are new assignments. This forward chaining may continue in several steps until the new state is reached. Again, processes must be able to unambiguously refer to the state at any time, so each slot in the rule-layer executive is a data structure that contains the new and old slot values as well as the time for the update. In a sense, we have adopted an object-oriented view of the rule base because we associate a set of rules with each slot. This makes the system more flexible and easier to maintain.

We have implemented two versions of RLX. The Common Lisp version runs under Unix as a development environment; the C version runs under pSOS$^+$ and is used for real-time execution. In both versions, rules and slots are automatically derived from definitions in a common language.

Interlayer communication. A piecewise constant fluent, the *communication fluent*, models the information flow between the process, rule, and analysis layers. The communication fluent relates the output fluent from the sending layer to the input fluent of the receiving layer.

The communication fluent is implemented as a message-passing system. Each time there is a discontinuity in the fluent of the sending layer, the message-passing system prepares a message to be sent to the receiving layer. Each message specifies a change in the value of the receiving layer's fluent. Messages from the process layer are also time stamped, which forces the receiving layer to treat updates in the same period as concurrent. The value of the attached time stamp is determined by the number of flips performed on the fastest subvector.

SATISFYING REAL-TIME DEMANDS

Our framework captures application time demands and transforms them to time constraints on different layers and on the entire application system. The modularity and simplicity of our implementation tools makes it easy to verify that computations will meet their deadlines.

Time and clocks. The estimator, the adapter, and PLX use a real-time clock to synchronize sampling and periodic computations that involve sampled data. The shortest sampling period, corresponding to the period of the fastest subvector, provides a clock that affects activities in the entire system. Each tick of the clock corresponds to a flip in PLX.

The clock in the process layer is used to label consecutive states, manage flip intervals for subvectors, and time-stamp assignments sent to the higher layers.

The rule layer is oblivious of time. The time stamps associated with its input fluents are used merely to partially order recognized events.

The analysis layer treats time stamps from recognized events as both a sequencing mechanism and a mechanism for reasoning about the time between events. Consequently, it must be aware of the global time between two ticks.

A large part of verifying time constraints is knowing the layers' worst-case execution times.

Time-constraint verification. For an application in the process layer, we can determine at compile time whether the computations to be performed in each cycle will meet their deadlines. Verification has the following steps:

1. Specify characterization routines and the application's time demands. We have devised a specification language that lets designers formally specify these two elements.[5]
2. Verify that the specification meets an application's time demands. We plan to identify verification procedures for the specification language and develop tools to check if the specification meets these demands.
3. Translate the specification to the language used in the process layer.
4. Check that PLX will perform computations in each cycle within the given time. The Sampling Theorem — which determines the smallest sampling rate required for sampling a signal without loss of information — dictates that the specification's upper bound is T on the shortest cycle for the application program. Designers merely check if PLX can perform its computations in time less than or equal to T.

A large part of time-constraint verification is knowing the worst-case execution time of the process, rule, and analysis layers.

PROMETHEUS: THE ULTIMATE BACKSEAT DRIVER

Prometheus is a large European cooperative project for developing automotive information technology. It started in 1987 at the initiative of automobile manufacturers in Germany, France, Great Britain, Italy, and Sweden and also involves electronic suppliers, universities, research institutes, and government agencies.

The overriding goals of Prometheus are to enhance the safety, convenience, and economy of automobile driving and to reduce its environmental impact. Work includes the development of route-guidance systems, car-to-car communication, codriver computer systems, and new sensor technologies. Implemented systems and prototypes were demonstrated by Prometheus participants in Torino, Italy, September 1991. Important next steps are to develop a common system architecture to support integration and to assess safety, effectiveness, and acceptance in field trials.

Our contribution to Prometheus is a high-level software architecture for the onboard system. The architecture must provide a framework for software that receives input from sensors, other cars, and the roadside information system (which might provide traffic information for a metropolitan area, for example) and use that input to build and continuously revise a model of the surrounding traffic. Work on the software architecture has been closely tied to work on standardized data formats and high-level protocols for dataflow among software modules — what are called fluents in the main article.

The codriver system is ultimately expected to provide a considerable number of services and to operate in several modes, from warnings of forthcoming obstacles, up through emergency warnings, to possible direct intervention to avoid a collision. In a longer perspective, all these services will probably be based on a model of the traffic situation surrounding the driver's car. The codriver system may also include a model of the driver's present knowledge about the situation and of the intentions for travel that the driver has communicated to it.

Several groups participate in Prometheus research and development in these areas. The work has been coordinated through a European committee for ProArt (the AI branch of Prometheus) and through the high-level Systems Engineering Group.

For more information on Prometheus, contact Prometheus Office c/o Daimler-Benz, Dept. TEA/F, PB 60 02 02, D-7000 Stuttgart 60, Germany.

REFERENCES

1. H. Glathe, "Prometheus — Safe and Effective Mobility in Europe," *Proc. Intelligent Vehicle and Highway Systems*, 1992 (available from the Prometheus Office).

Process layer. We have determined the PLX overhead — the extra CPU cycles needed to synchronize tasks, manage memory, and communicate between layers.[6] This overhead is a function of several application-dependent parameters derived from the PLCL description. The computed overhead, together with the upper bounds on the worst-case execution time of the library modules, forms the basis for analyzing the worst-case execution time of the process-layer application.

We have tested our framework and tools on a number of applications, mostly in traffic safety.

Rule layer. Given a set of slots and rules in the rule layer, designers can make a similar check to determine the maximum response time from firing rules. Because rules are organized around specific slots, each slot has a finite number of rules attached to it, and the range of each slot's values is finite. Thus, we can determine (statically) the maximum number of inferences after each slot value changes at runtime.

Al Mok[7] has studied the real-time aspects of rule-based components. We have adopted and extended this work by empirically analyzing the behavior of the rule layer as part of one integrated prototype. During this work, we discovered the need for a tool that would automatically verify maximum inference duration.

The rule layer should react to a given assignment sent from the process layer within the amount of time allowed by the application for that reaction. We derive the maximum (allowed) response time for the rule layer by subtracting the time needed by the process layer for characterization and control and two (worst-case) interlayer communication delays from the total response time imposed by the application. We must verify that the remaining time is sufficient for rule-layer computations.

Analysis layer. We envisage a similar requirement on the analysis layer. When reasoning is to guide the choice of existing rule sets in the rule layer in response to a sequence of events in the environment, an application-bound, hard deadline is also in the analysis layer. However, the analysis layer may carry out computations with a soft character, as well. For example, it may record the history of its decisions and use a learning mechanism to optimize the rule sets in the rule layer. This activity is not subject to hard deadlines, and efficient computations are adequate. In this case, a rule set should be suggested only if its maximum inference duration does not exceed the available reaction time in the rule layer.

TWO APPLICATIONS

We have tested our framework and tools on a number of applications, mostly in traffic safety. In two applications — an elevator-control system and a driver-support system — we evaluated

- ♦ how to integrate components with different functionality running on distributed machines,
- ♦ the performance of the communication package, and
- ♦ how our mechanisms to specify and validate time constraints were working.

Elevator-control system. In this application, the system controls the behavior of a miniature elevator located in our laboratory. The (physical) level of the elevator is continuously measured by a height sensor connected to the process layer. A control panel and the motor are also connected to the process layer.

Our main reason for implementing this system was to get experience from integrating the process and rule layers. Even though this example is simple, it involves all the fluent transformation types. It also provides a simple example of a multilayer path: The characterizer receives the current level from the estimator; if the level has become equal to the desired level, a message is sent to the selector. The selector then realizes that the elevator should go into stop mode, and sends a message to the effector, whose control algorithm is changed.

The results of this application showed us that we could combine the process and rule layers as intended with an adequate response time.

Driver-support system. In this application, the system supports a driver during passing and lane merging. The system warns the driver when predefined safety criteria are not being met in an environment containing an arbitrary number of other cars. This application is also implemented by modules in both the process and rule layers.

The characterizer performs transformations on quantitative sensor information about each car (such as giving the distance to the driver's car) and arrives at more abstract models of the environment. The system represents cars of interest by deictic reference — in this case, symbolic locations relative to the driver's car. Thus, the system refers to "first car in the left lane ahead of driver's car" instead of "the red Volvo with license number X."

The system recognizes the properties and trends of individual cars by monitoring a collection of quantitative values against thresholds with some hysteresis (the system uses two thresholds, one for rising and one for falling signals, to be more robust against noise). An example is computing — for each surrounding car — a distance factor as a function of the speed, acceleration, and position of the driver's car and the other car. The distance factor is then converted to a qualitative closeness value like "too close."

The system uses the properties of groups of cars to recognize significant events, such as a blocked left lane. On the basis of these events and the rules in the selector, the system updates the rule-layer model and, if appropriate, selects a warning and sends it to the effector. The selected warning is then output to the driver through a text panel or voice channel.

This application illustrates how the rule layer is used naturally to handle complex decisions on the basis of recognized events. The shortest cycle in the process layer for this application is 10 ms.

Because we do not yet have access to raw sensory information about all surrounding cars, we have had to rely on sensor data produced by a simulation module. However, we believe that system performance for this application will not differ greatly when we have actual data. We have received real sensor data about one car derived from physical sensors and a camera mounted on the car. PLX easily coped with the computations involving regular fluents for the output of this one car. We are confident that adding more cars would not affect system performance because the simulated data we have fed to the system would be the same type as the real sensor data with many cars.

Furthermore, traffic researchers studying the frequency of conflicts in traffic and normal reaction times have found that the elapsed time from a conflict's first appearance to the avoidance of an accident is often only a few seconds. Thus, we can expect to resolve at least certain types of conflicts if the system makes the appropriate decisions within a half to one second.

The architecture we have described is suitable for integrating symbolic and numeric computations in a range of applications. The implementation tools support the types of computations necessary in the process and rule layers.

We have outlined the functional characteristics and time constraints of the analysis layer, and we are working with time formalisms to be used for eventually specifying software in that layer.

We also plan to develop tools implementing applications in the analysis layer — including a mechanism that will let it use the static analysis methods at lower layers for arriving at runtime decisions.

Several applications that require reasoning in dynamic environments have been developed using this framework. In addition to our work on tools for the analysis layer, we plan to devise verification and static-analysis tools for the process and rule layers that give designers assurance at compile time that the application's time demands will be met. ♦

ACKNOWLEDGMENTS

We are grateful to Jacek Malec for contributions to the work described here and for comments on earlier drafts of this article. This work was supported by the Prometheus project within the IT4 research program, and by the Center for Industrial Information Technology.

REFERENCES

1. H. Verbruggen and K. Åström, "Artificial Intelligence and Feedback Control," *Proc. IFAC Workshop on Artificial Intelligence in Real-Time Control*, Pergamon Press, Oxford, 1989, pp. 1-11.
2. R. Brooks, "Intelligent Without Reason," *Proc. Int'l Joint Conf. Artificial Intelligence: Volume 1*, Morgan-Kaufmann, San Mateo, Calif., 1991, pp. 569-595.
3. E. Gat, "Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots," *Proc. Conf. on Artificial Intelligence*, MIT Press, Menlo Park, Calif., 1992, pp. 809-815.
4. J. Hultman, A. Nyberg, and M. Svensson, "A Software Architecture for Autonomous Systems," *Proc. Int'l Symp. Unmanned, Untethered, Submersible Technology*, University of New Hampshire, Durham, 1989, pp. 279-292.
5. S. Nadjm-Tehrani and P. Österling, "Characterization of Environment Conditions with Metric Temporal Feature Logic," *Proc. IEEE Int'l Conf. AI, Simulation and Planning in High-Autonomy Systems*, IEEE CS Press, Los Alamitos, Calif., 1992, pp. 267-274.
6. M. Morin, "Performance Aspects of the Process Layer Executive," Tech. Report LAIC-IDA-91-TR13, Linköping University, Sweden, 1991.
7. A. Mok, "Formal Analysis of Real-Time Equational Rule-based Systems," *Proc. IEEE Real-Time Systems Symp.*, IEEE CS Press, Los Alamitos, Calif., pp. 308-318.

Address questions about this article to Morin at Linköping University, Dept. of Computer and Information Science, S-581 83 Linköping, Sweden; Internet peros@ida.liu.se; Bitnet peros@seliuida.

Magnus Morin is a PhD candidate in computer and information science at Linköping University in Sweden. His research interests include architectures and languages for real-time software and operating systems.

Morin holds an MSc in electrical engineering from Linköping University.

Simin Nadjm-Tehrani is a PhD candidate in computer and information science at Linköping University in Sweden. Her research interests include logic programming and the use of temporal logic in specifying and verifying safety-critical systems.

Nadjm-Tehrani holds a BSc in computer science and accounting from Manchester University, England, and a Licentiate in computer science from Linköping University.

Per Österling is a PhD candidate in computer and information science at Linköping University in Sweden. His research interests include the modeling of dynamic environments and the use of temporal logic in specifying and verifying real-time systems.

Österling holds a BSc in computer science from Linköping University.

Erik Sandewall is a professor of computer science at Linköping University in Sweden. His primary research interests are knowledge representation for time and action, and non-monotonic reasoning. He was the European coordinator for Prometheus's ProArt for several years.

Sandewall holds a PhD in computer science from Uppsala University, Sweden. He is a fellow of the AAAI.