An Integrated Software Environment for Development of Driver Support Systems

J. Malec, R. Farahani, M. Morin, S. Nadjm-Tehrani,
P. Österling, and E. Sandewall
Department of Computer and Information Science
Linköping University
S–581 83 Linköping, Sweden
e-mail: [jacma, rezha, magmo, simin, peros, erisa]@ida.LiU.se

April 15, 1996

1 Introduction

Our contribution to the development of reliable driver support systems is based on a methodology which provides the possibility of systematic validation of individual software components before integration into a complete system. The methodology is closely linked with a series of software tools which together form an environment in which we have now implemented the first support prototypes.

The overall approach and the initial implementation efforts for parts of the environment have been reported earlier. This paper gives a summary of how the integrated environment was used to support the development of a support system which provides information and warnings to simulated cars in a variety of traffic situations.

The software components form a layered architecture whereby the input data is analyzed and classified in successive steps. First by fast low level routines in a level denoted by the *process layer*, and then in a rule-based environment called the *rule layer*. The software modules included in the integrated environment consist of:

- a *sensor simulation package*, providing pre-recorded sequences of sensor values for the own car and a variable number of cars in the environment.
- a *process layer executive* [MS90], supporting the real-time processing of the pseudo-continuous recorded data in the input streams, where the aim of processing is extraction of certain characteristics of the dynamic input stream for reporting to the rule layer.
- a *rule layer executive* [San90a] which supports the identification and classification of interesting or critical traffic situations based on the rules provided in a knowledge base.
- a number of *communication* routines [Öst91] for linking the above components through sending pseudo-continuous or discrete data flows.
- a *road segment data base management module* [Far91], which creates and maintains a road map data base, as well as providing an interface to the database for application programs.

The unifying concept between the various software modules is the concept of *fluent* as an abstraction of time-dependent data flow. Each part of the environment can be seen as a tool facilitating particular types of transformations on fluents. This is due to the fact that the application itself, i.e. the collection of programs which provide driver support in a dynamic environment, can be seen as transformations on fluents.

Therefore, in what follows we briefly describe the types of transformations involved in order to set the scene for describing each software tool or the developed prototype.

We associate *fluents* - functions whose values vary over time - with each interesting parameter in the environment. We particularly distinguish between *continuous* fluents - that is fluents representing piecewise continuous functions, and *discrete* fluents whose values are, in principle, discrete and piecewise constant with occasional discontinuities. In order to get a time discrete representation suitable for handling in a computer, continuous fluents can be approximated by sampling at fixed short intervals. The approximated continuous fluents will henceforth be referred to as *periodic* fluents. Since discrete fluents are piecewise constant they can be realized as flows of messages which occur asynchronously.

Computations can be viewed as transformations on fluents. Consequently, we can distinguish between four classes of computations:

- periodic fluent to periodic fluent. An example could be the transformation performed by a simple control system. Such a system could be described as a transformation on a fluent whose values are sensor signals into a fluent of actuator outputs where the transformation is defined by the implementation of the regulator.

- periodic fluent to discrete fluent. Such transformations may result from discontinuities in the periodic fluent causing a discontinuity in the discrete fluent.

- discrete fluent to discrete fluent. One such example is discrete event simulation.

- discrete fluent to periodic fluent. These typically result from a discrete fluent influencing the control algorithm of a periodic fluent.

In [San90b] we grouped different types of transformations in different blocks. The estimator performs sampling and preprocessing of sensed continuous fluents. The characterizer classifies the current situation and detects discontinuities. The effector implements the basic control algorithms, and the selector determines the control algorithms to be used based on characterizer output.

Our current prototype processes the simulated sensor information [Far91] by characterizing the current situation [NTÖ91], sending the relevant discrete messages to the rule layer, and produces a number of information and warning messages based on further processing of the information in the rule layer. The system provides support to the own driver in a variety of traffic scenarios. These include approaching the curves in the road with different speeds, overtaking the car in front in presence of an arbitrary number of cars in the environment, approaching and passing different types of intersections in presence of other cars, etc.

The paper is organized as follows: in section 2 we give a brief description of the components making up the environment. We then go on to describe how the first prototype was embedded in the environment, and give an account of the implemented support functions. Finally, a summary of the work is then presented in section 4.

2 The Integrated Software Environment

2.1 The Sensor Simulator Module

In order to provide stable inputs for repeatable tests of the driver support system software, a graphical sensor simulation tool has been developed. The task of the simulator is to create a periodical fluent containing the outputs (e.g. position, signals) of the sensor system to be simulated. Note that we are only concerned with the outputs of the sensor system, not with how the results are obtained.

Each run of the simulator generates a file of simulated sensor data for one car. Scenarios with several cars are created in an incremental fashion. Previously recorded cars are reused by the simulator to represent the presence of other cars.

The simulator is interactive in the respect that the user controls the car that is currently being simulated by pressing keys on the keyboard at the same time as the simulated car and its environment are animated on a graphical display.

2.2 The Process Layer Executive

The process layer executive facilitates the implementation and maintenance of the hard real-time parts of the application which performs the transformation 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) 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) [Mor91b] is used to define the components of the dual state vector and to configurate 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 the application [Mor91a].

2.3 The Rule Layer Executive

The rule layer executive facilitates the rule-based discrete fluent to discrete fluent transformations of the selector block. The software is therefore referred to as the *Discrete Fluent Manager*.

In the rule layer, the fluent is represented 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 *Discrete Fluent Manager* 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 (cf. Section 2.4). 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.

2.4 Inter-layer Communication Package

The information flow between the layers is characterized by a discrete fluent which determines the value of a set of slots (in the other layer).

The nature of the fluent enables us to reduce the communication between the layers to messages indicating that a discontinuity has occurred in one of the slots. Consequently, we have implemented the fluent as a flow of messages. Each message is a statement representing a change in a slot value, i.e. an *assignment*.

The fluent exists at three locations of the software architecture, namely:

- In the process layer;
- In the inter-layer channel;
- In the rule layer.

Different representations of the fluent are needed in the different locations as well as transformations from one representation to another.

In the process layer, the fluent is represented by a set of parameters in the dual state vector. When the value of one of the parameters changes, a time-stamped assignment is prepared and sent to the inter-layer channel. The value of the attached time-stamp is determined by the number of flips performed. The inter-layer channel transmits assignments between the layers. The internal representation of an assignment in the inter-layer channel depends on the medium used for communication, e.g., shared memory, serial link, ethernet. In the rule layer the assignment is received and the forward chaining initiated.

Communication from the rule layer to the process layer is handled in a similar fashion. A discontinuity in a a slot of the rule layer fluent causes an assignment to be prepared and sent to the process layer through the inter-layer channel.

2.5 The Road Segment Data Base Manager

The *Road Map Data Base* contains all static information about the road network that is needed in the driver support system. A road map is represented as a hierarchical data structure, which on the top level consists of intersections and streets connecting them.

For efficiency, the road map must be manipulated in primary memory. However, in examples of realistic size the road map has to be subdivided into partitions that are small enough to be loaded into a primary memory map cache. The *Road Map Data Base Management System* is responsible for keeping a partition of the road map which covers the immediate surroundings of the own car in the cache. As the car moves around, the contents of the cache is updated accordingly.

3 Implemented Support System

In this section we set out the methodology for separate development of application components prior to their integration. As a working example we have chosen the prototype driver support system presented at the Board Members' Meeting 1991[1].

In order to provide a meaningful and timely support to the driver, either as a help message, or as a warning, a driver support system (DSS) has to acquire, possess and use knowledge about the following:

1. the environment in which the car finds itself. It may be further divided into two parts:
    (a) knowledge about *static* parts of the environment, as e.g. the road network, for each road the number of lanes and their sizes, curvature of the bends, fixed speed limits (imposed by traffic signs), areas of increased probability of accidents (schools, shopping areas, bicycle path crossings, etc.);
    (b) knowledge about the *dynamic* element of the current traffic situation, i.e. configuration of cars in the proximity of the car, setting of traffic lights, existence or absence of obstacles on the current lane, rush hours, etc.;
2. the current state of the car (its position with respect to some coordinate system, direction, speed, acceleration, state of the safety-related elements, etc.);

[1]Torino, September 1991, CED–3.

Figure 1: The simulated road environment.

3. the estimation of driver's intentions or plans. Especially important is the assessment of the driver's awareness of degree of "danger" which may be attached to the current situation;

4. the set of possible maneuvers of the car, and their utility in current situation; and finally

5. the set of possible help/warning messages, their utility, and expected driver's reaction to them.

The prototype implementation of DSS we are going to describe here covers, although not in the full extent, all the topics mentioned above. Because we did not have access to sensor data gathered in real traffic situations, we have used a simulated set of sensor outputs for a road environment consisting of several straight road segments, bends, and intersections (cf. Figure 1).

In this environment we simulate movements of three cars: one usually described as the OWN car, denoted by a rectangle around the mark, and two other road users. The information about positions of all the cars, plus other important information about the own car is then input as simulated sensor data to the process layer of the DSS. This way of providing the input data allows us to adjust to any possible kind of input sensory data, independently of its source. We can imagine using processed vision information, messages sent via radio link from either environment, or the other road users, low-level sensory information about both the car and the environment, etc.

The process layer is equipped with the road-segment database encoding the "static" part of environment description. The "dynamic" part is formed from the preprocessing of input data, and is stored as the state vector consisting of all the state variables required to provide meaningful support to the driver. Some of the state variables, as e.g. the classification of the configuration of cars in the current situation, or the last moment to slow down before some dangerous area such as an icy bend, require non-trivial amount of processing to be performed in real time.

One of the most computationally demanding, and in the same time most frequently used processes is *coordinate transformation*. As the input informing about the own car's position we take the incremental data providing difference between previous and current positions of the own car, with reference to the road segment the car is currently on (in other words, the position is expressed in road-segment coordinates). On the other side, distances between the trafficants are computed using the Cartesian coordinates (and sometimes even collected in this form). Therefore each time new data about the own car's position is obtained, it must be recalculated into Cartesian coordinates, and then for all other trafficants back to road-segment based ones. Due to hard real-time constraints this task must be as efficient as possible, and therefore it has been given much attention during implementation of the DSS prototype.

The configuration of transformations which have to be implemented in the process layer, and the data paths between them, can be specified using the Process Layer Configuration Language. The actual code of the processes is implemented in the C language.

The information relevant to the decision to be made by the DSS (whether to act, and how to act) is sent to the rule layer of the system. Examples of such relevant pieces of information are: lane changes by the own car, changes of traffic condition in each lane in terms of degree of freeness during the overtaking maneuver, changes of the speed limit. Some of the slots in windows presented on Figure 2 are just reporting current values as seen by the rule system.

The information about both how to act in different traffic conditions, and how to support the driver in case he is not acting according to the prescribed rules of behavior, is encoded in the rule layer using a set of rules. The intentions of the driver are currently recognized from the set of observed maneuvers (e.g. lane changes, accelerating, braking), and some additional data (as blinkers of the own car being on). If the driver's behavior apparently differs from the expected one, and if this may lead to dangerous situation, the decision of issuing a warning message is taken, and is sent to the driver. In our implementation these messages appear in fields of the windows presented in Figure 2.

For example, if the current situation with respect to overtaking is classified as unsafe (the driver is trying to overtake the car in front in spite of presence of the meeting car), this information is passed to the driver, with possibility of increasing urgency of the messages ("soon-unsafe", "unsafe", "emergency"). The same applies to speed-related warnings. When the DSS has decided that the driver should have started slowing down before a speed limit (due to a bend, or a sign, or some other conditions) it increases the intensity of warning ("soft" meaning you should start slowing down now not to drive uncomfortably, "firm" meaning "brake firm, or you'll be in big trouble", and "alarm" meaning "you ARE in a big trouble already" — this might correspond to some intervention in control taken by the DSS itself, as e.g. braking).

The set of rules to be encoded in the rule layer is produced in two stages. First, the set of possible traffic conditions is specified using Process Transition Networks formalism. The intended ("model") behavior of the own car is then specified, using the same tool. Finally, the support actions are encoded for various dangerous situation the driver may encounter. The PTNs are then translated into a rule-based logical language, which the rule layer interpreter

Figure 2: The windows of the rule layer.

can understand. After checking the consistency of the set of rules, they are finally coded in the system.

4 Summary

A driver support (co-pilot) system is a typical example requiring distributed computations in real time. The use of different computing resources and the nature of performance constraints on the sub-components dictates solutions where individual components *operate* semi-independently. But a satisfactory integration of the sub-components is only possible if there is a unifying framework already at the design stage.

The different nature of tasks performed by each component in turn allows the *development* of the components in an independent fashion. Hence, there is a need for definition of connection interfaces, and a methodology for development which facilitates systematic generation and pre-integration testing of individual components.

We have proposed the use of a layered software architecture for the Prometheus driver support systems. The development of prototypes for such support systems requires a number of tools each supporting a particular type of computation. In this paper we have described the tools implemented within our group, and indicated the unifying framework for the independent tools using the concept of *fluent*. The results of this approach are promising. We have developed a methodology which provides the possibility of systematic validation of individual software components (process layer's processes, rule layer's behavior specifications, and possible support activities) before integration into a complete system.

We have also described the first version of a support system, developed within the aforementioned environment. The system monitors the input sensory information for the own car and a number of cars in the environment, and performs necessary computations arriving at measurements relative to a predefined map in its infrastructure data base. The quantitative information is then transformed into qualitative descriptions about the current scenario, building a model of the environment by the characterizer component. The recognized significant events are then reported to the selector module which determines what messages are to be written on information and warning display units. Initial experiments allow us to expect that the real time constraints will be satisfied.

The operation of the support system within the environment was demonstrated at the Board Members' Meeting in Torino in 1991.

References

[Far91] Reza Farahani. A Road Segment Database Management System and Simulation Tools for a Map-based Copilot. In *Proceedings of ProArt Workshop on Intelligent Co-pilot*, Grenoble, France, December 1991.

[Mor91a] Magnus Morin. Performance Aspects of the Process Layer Executive. Technical Report LAIC-IDA-91-TR13, Linköping University, 1991.

[Mor91b] Magnus Morin. PLCL – Process Layer Configuration Language. Technical Report LAIC-IDA-91-TR10, Linköping University, 1991.

[MS90] Magnus Morin and Erik Sandewall. A software architecture supporting integration of sub-systems. In *PROMETHEUS: Proceedings of the Fourth Workshop*, 1990. Also available as Technical Report LAIC-IDA-90-TR21, Dept. of Computer and Information Science, Linköping University.

[NTÖ91] Simin Nadjm-Tehrani and Per Österling. Characterization of Environment Conditions as a Basis for Providing Driver Support. In *Proceedings of ProArt Workshop on Intelligent Co-pilot*, Grenoble, France, December 1991.

[Öst91] Per Österling. The Inter-Layer Communication. Technical Report LAIC-IDA-91-TR7, Department of Computer Science, Linköping University, 1991.

[San90a] Erik Sandewall. Discrete Fluent Management System, a program in the HIRW Workbench. Unpublished report, 1990.

[San90b] Erik Sandewall. System Block Specifications and their PROMETHEUS applications. In *Prometheus: Proceedings of the Fourth Workshop*, 1990. Available also as LAIC-IDA-90-TR22.