A flexible application framework for distributed real time systems with applications in PC based driving simulators

M. Grein, A. Kaussner, H.-P. Krüger, H. Noltemeier

IZVW (Center for Traffic Sciences at the University of Würzburg)
Roentgenring 11
97070 Würzburg, Germany
e-mail: grein@psychologie.uni-wuerzburg.de

Abstract

For the research at the IZVW (Center for Traffic Sciences at the University of Würzburg) a driving simulator was build, that consists of 11 personal computers with a 180° front projection system, 3 rear projections and a 6-6 stewart platform. Also, three other driving simulators consisting of one, two and five personal computers are used at the IZVW for different experiments. One desired goal of the driving simulator software is, that the software framework would allow to adapt very felxibly to the most different experiments. A special aim was to easily integrate different driver support systems. Also the software must ensure, that the experiments can be performed unchanged on each simulator as far as hardware resources allow it. These goals forbid hard-coded connections of the single parts of the simulator software, e. g. vehicle dynamics simulation and driver interface. In this article, we give an overview of the software framework of the IZVW driving simulators.

Introduction

The research done at the IZVW in the domain of traffic science requires a flexible software concept for covering all applications of driving simulation. At the moment, four simulators with differnet architectures are used: a mobile system consisting of a single personal computer and a testing and development simulator driven by two personal computers. It is steered by a joystick. Furthermore there is a static simulator with a simple mockup and one front channel projection system. The most complex simulator consists of a 180° front projection system and 3 rear projections. It also has a 6-DOF motion system (see Figure *1*). This simulator is controlled by 11 personal computers.

Figure 1:IZVW driving simulator

The software of the simulators is required to run on these four architectures and must ensure that the experiments can be performed unchanged on each simulator as far as hardware rescources allow it. Moreover, the software must be augmentable by additional modules (e.g. driver support systems) without compiling it as a whole. In this article we give an overview of the software developed as a framework for distributed real time systems like the IZVW driving simulators.

Overview

Basically every driving simulator consists of program modules that have to perform specific tasks independently. The driver interface provides data like steering angle, position of accelerator pedal and brake. Through this interface the driver is informed about velocity and steering forces. The vehicle dynamics simulation requires these outputs to compute the physics of the car that moves in the scenery database. If the simulator has a motion system the car's accelerations control the motion cueing. The inputs for sound simulation come from the vehicle dynamics simulation and the vehicles in ambient traffic. Their behaviour in turn is computed by a module called traffic. The visualization system renders images of the scenery database and the traffic scenario.

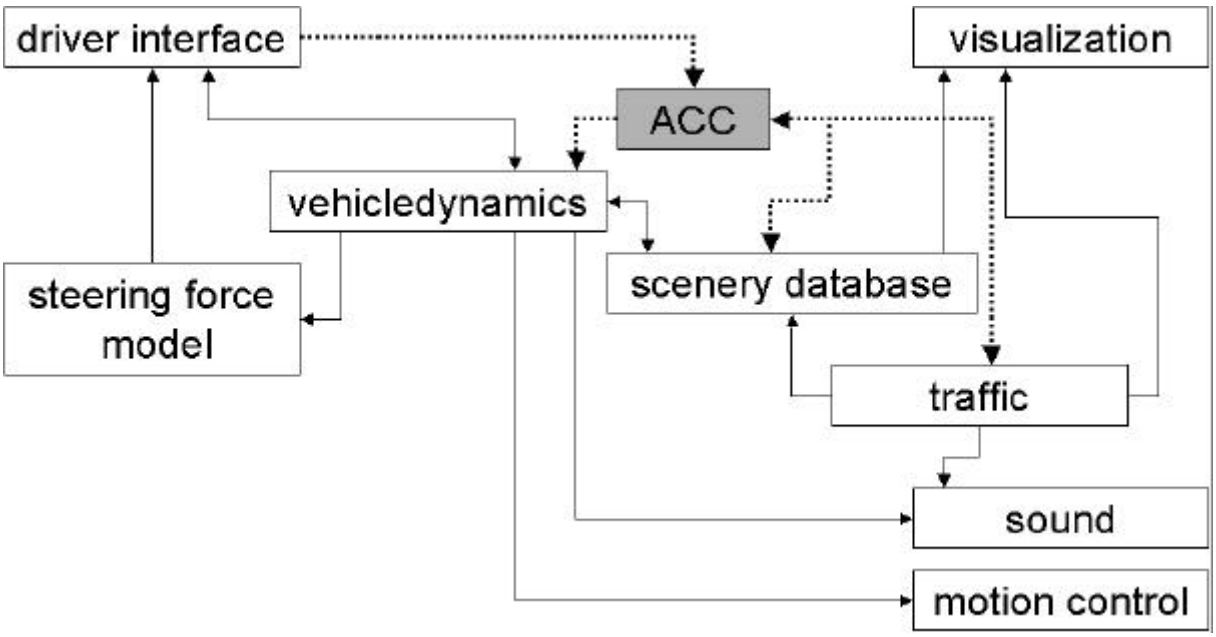

Figure 2: program modules of a driving simulator

If there has to be integrated a specialized module for experiments the original configuration of the software must be changed. For example, an ACC system is plugged in between the driver interface and vehicle dynamics simulation cutting the former connections of acceleration pedal and of brake. Additionally the ACC system needs information about the vehicle in front like distance and velocity.
Since at the IZVW many experiments are carried out concerning different combinations of such specialised modules the connectivity of the modules must not be hard-coded. In fact the configuration is determined individually for each experiment by the researcher.
The software framework presented in the following is able to fulfill these requirements.
Figure *2* shows an overview of the components of the framework.

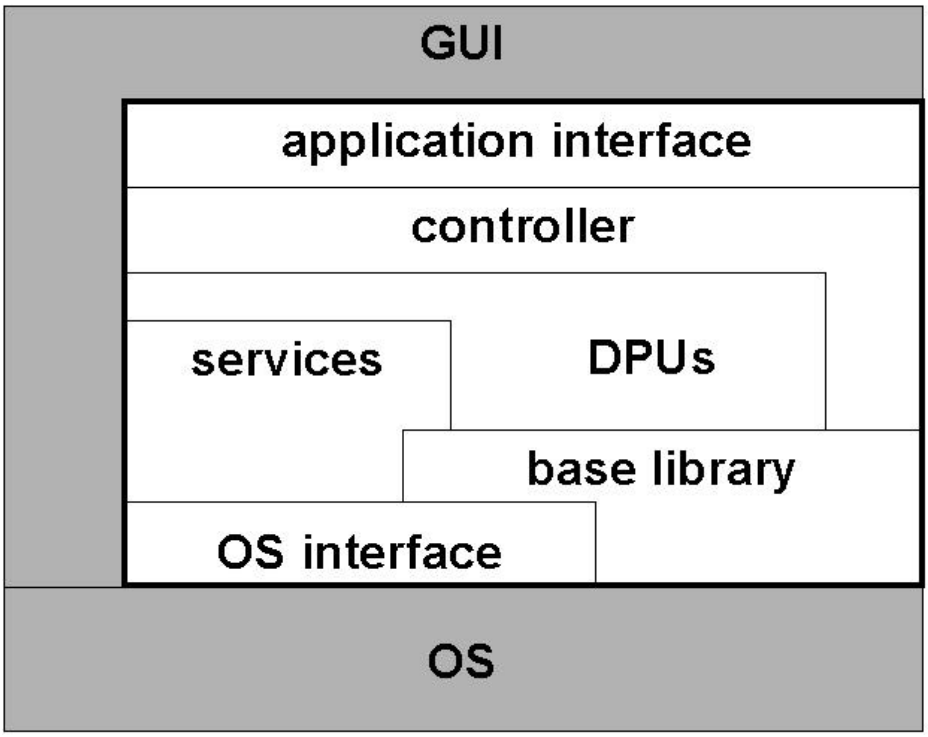

Figure 3: components of the framework (white)

The above mentioned modules of the simulation are encapsulated in so called DPUs (Data Processing Units). The framework can load them at runtime, assign them to responsible computers and connect the respective inputs and outputs of different DPUs. All this information is defined in a configuration file.
The controller supervises the state and the timing of the DPUs. It also organizes exchange of data between the computers so that the developer of a DPU does not have to worry about the communication with other DPUs. The application interface facilitates the developers access to the controller and guarantees its consistent use. For example, it serves as an interface for GUIs (graphical user interface) through which the simulation is used by the researchers (see figure 3).

Several services are provided by the framework. They are used by the DPUs if information sources or hardware resources have to be shared. To run the framework on different operating systems, all OS specific funtions are implemented in the OS interface. This has the effect that only this library has to be rewritten for other operating systems.
The base library provides frequently used data structures, several base classes (e.g. for DPUs), networking classes and so on.

DPUs

Data Processing Units determine the functionality of a driving simulator. They are derived from a common base class and therefore have a common interface to the controller. This interface has commands for starting the DPU, performing a single computation step, pausing or stopping the DPU. Each DPU is implemented in a separate shared library so the controller can load them at runtime depending on the actual configuration file. Thus, a simulator can be upgraded with new functionality by simply uploading a newly developed DPU without changing the system as a whole. After beeing instantiated by the controller the DPUs register

their input and output variables in the controller. The controller makes these variables available for each component in the simulation. The variables are accessed via their names. By convention, related DPUs always have the same set of names for their variables.

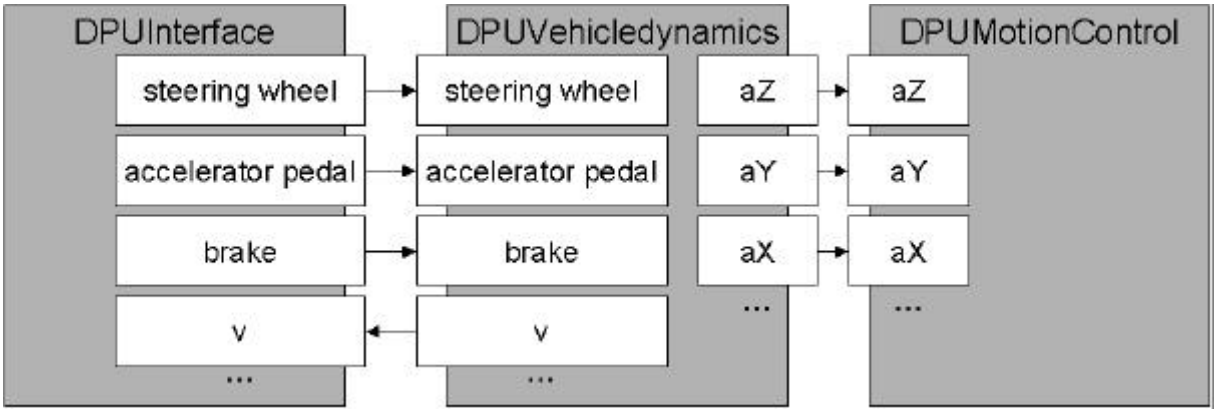

Figure 4: a simple DPU connection

For example the DPU implementing an interface to a completely equipped mockup has at least the outputs steering angle, accelerator pedal and brake as well as inputs like velocity, engine speed and steering force. A DPU that implements an interface to a joystick provides the same input and output variables even though velocity, engine speed and steering forces have no effect. By doing so, the two DPUs may easily be exchanged.

Controller

Each computer in the simulator runs exactly one controller. The controller may work in two modes:

(1) in the operator mode: the computer used by the operator starts, pauses and stops the simulation and manages the controllers on the other computers.
(2) in the remote mode it is controlled by the operator controller from which it receives all commands.

At the beginning of the simulation the operator controller reads the configuration file, starts the respective remote controllers and awaits their checkback signal. Afterwards each controller loads the configuration file and instantiates its DPUs. As mentioned above, in this state the DPUs register their variables. In the next step each controller has to connect the inputs and outputs of the DPUs as described in the configuration file. Firstly each controller connects the variables as far as it can be done locally. Variables that are not locally available are requested from other controllers via the network. For safety reasons the operator controller supervises the remote ones using a watchdog system. The remote controllers perform a handshake with the operator controller at a certain frequency. If the handshake procedure is interrupted in either direction the simulation is halted.

Sometimes it is not possible to encapsulate third party modules in DPUs. In this case the operator controller provides several network communication channels. Using the controller state channel an extern application can query the state of the simulation system. The data channel offers access to all variables of DPUs registered in all controllers. Thus the variables can serve as input for external modules and DPU parameters can be adjusted by them.

Services

So called services handle the information that is used by different modules. The services also manage hardware resources that are shared by different components.
The following example shows how to integrate the scenery database architecture described in [1] into the framework using DPUs and information services. On the one hand, the scenery database delivers information like road gradients and road roughness to the vehicle dynamics simulation. On the other hand, it provides rendering information (like vertices of the road network and object positions) for the visualization system. For that reason the scenery database is implemented as a service which is accessed by the DPU representing the vehicle dynamics and by the visualization system service. In turn, this service is used by the DPUView for rendering the scenery seen from a certain camera position. Every DPU can insert graphical objects into the visualization system service and animate them. Since there is complete access to the vertices of the objects complex animations can be realized. Another example for an information service is the ambient traffic from which a DPU can query position and behaviour of any road user.

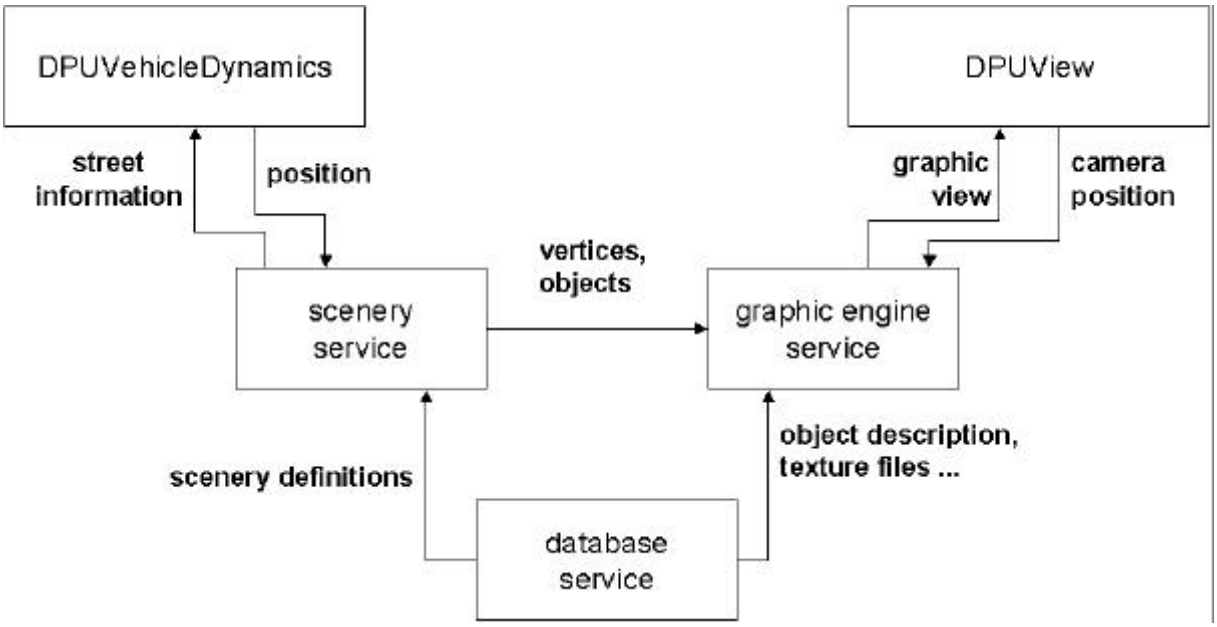

Figure 5: DPUs and different use of services

The second task of services is to manage the access to limited hardware components (e.g. channels of a soundcard). The hardware service has no direct access to the OS. Therefore, functions which are dependent on the OS must be called by the service via the OS interface.
Several parts of the framework (controller, visualization system, sound system) use a specific service that provides access to a central database containing graphical objects, textures, configuration files, sound samples and binaries of system libraries and DPUs. If one of these files has to be opened the database service contacts the database server and downloads the actual version. Therefore, the whole system can be centrally updated.

Configuration

The computers required for a specific experiment and their respective DPUs are described in a simple configuration language. The setup of a simulator for an experiment (apart from scenario definition) is defined in two parts of the configuration file.
The first part "ComputerConfiguration" consists of a pool containing all computers involved in the experiment. For each computer information like its IP adress in the network and the timing of computations is defined. Later on the computers are adressed via the names assigned in this section. For an example of a computer configuration see figure *6*.

```
Computerconfiguration
{
      Pool Simulator
      {
            Computer Operator
            {
                  IP = "10.1.2.150";
                  PeriodMS = 10;
                  Operator = true;
            };
            …
            Computer SimCar
            {
                  IP = "10.1.2.158";
                  PeriodMS = 2;
            };
            Computer Main
            {
                  IP = "10.1.2.158";
                  PeriodMS = 10;
            };
            Computer Graphics1
            {
                  IP = "10.1.2.159";
                  PeriodMS = 20;
            };
            …
      };
};
```

Figure 6: first part of a configuration file

In the next part one or more DPU pools are defined. Each DPU pool contains some DPUs and determines their connections. Before starting the simulation the operator can select one of these DPU pools. A DPU instantiated in a DPU pool has its own section where it is assigned to a computer and where the intial values of its variables can be set. The connectivity of the DPUs in a pool is described in the connections statement.

```
DPUConfiguration
{
      # First Pool with Basic DPU's
      Pool Basic
      {
            DPUDriverInterface Console
            {
                  Computer = {Simcar};
            };
            DPUVehicleDynamic Car
            {
                  Computer = {Main};
            };
            …
            DPUView VFront
            {
                  Computer = {Graphics1};
                  X=0;Y=0;W=1024;H=768;
            };

            …

            Connections =
            {
                  Console.SteeringWheel <-> Car.SteeringWheel,
                  Console.Brake <-> Car.Brake,
                  Console.AcceleratorPedal <-> Car.AcceleratorPedal,
                  …
            };
      };

      # Second Pool with ACC and all Basic DPU's
      Pool ACC : Basic
      {
            DPUAcc Acc
            {
                  Computer = {Main};
            };
            Connection =
            {
                  Console.Brake <-> Acc.Brake,
                  Acc.Brake <-> Car.Brake,
                  Console.AcceleratorPedal <-> Acc. AcceleratorPedal,
                  Acc. AcceleratorPedal <-> Car. AcceleratorPedal,
                  …
            };
      };
};
```

Figure 7: second part of a configuration file

A DPU pool can inherit information from another DPU pool (similar to classes in object oriented programing languages). This faciliates the use and easy maintenance of base configurations. For example, if an ACC system is integrated in a base configuration, a new DPU pool is derived, which instantiates the new ACC DPU and defines the modified connections. In the derived pool the signals of the accelerator pedal and the brake are not fed

directly into the vehicle dynamics simulation, but first were send to the ACC DPU. As a result the connections between accelerator pedal and brake defined in the base pool are cut automatically and replaced by the connections given in the derived pool (see figure *7*). The automatic cut down of the former connections is justified by the fact that it makes no sense to connect one input with multiple outputs.

Conclusion

The technical progress in the area of entertainment electronics allows the use of components developed for this market segment in driving simulators. Thus low cost simulators can be build that are scalable in performance and functionality. This requires a simulator architectures which allows that different software modules can be distributed on specialized hardware (sound, image generators, mockup interfaces, etc.). Their functionality must be easily configurable for each experiment. The presented software framework for distributed realtime systems can be used to run driving simulators that fullfill these demands.

References

[1] Kaussner, M.Grein, H.-P. Krüger, H. Noltmeier: An architecture for driving simulator database with generic and dynamically changing road networks, Proc. DSC 2001, Sophia-Antipolis, 2001
[2] Proceedings of the Driving Simulation Conference, DSC1999, Paris, 1999
[3] Proceedings of the Driving Simulation Conference, DSC2000, Paris, 2000