A versatile demonstrator for distributed real-time systems: Using a model-railway in education

Stephan Höhrmann Hauke Fuhrmann Steffen Prochnow Reinhard von Hanxleden
Christian-Albrechts University of Kiel, Department of Computer Science, Kiel, Germany
E-mail: {sho,haf,spr,rvh}@informatik.uni-kiel.de

***Abstract*— Teaching computer science in the domain of safety critical systems needs to address multiple topics: embedded and distributed systems, real-time behavior and model-based system design. We present a model railway demonstrator that employs these paradigms in one application. The distributed railway controller nodes are connected using industry standard networks (CAN, Ethernet or TTP). The corresponding application software can be implemented manually in C/C++ or by using model-based development tools. This flexibility enables users to compare advantages and drawbacks of various design patterns. Following an overview of the railway and its control system, experiences from educational courses as part of the DECOS training activities are discussed. In these courses university students were to implement controlling applications and simulations for the demonstrator using Matlab Simulink, Stateflow, and hardware specific add-ons.**

I. Introduction

Developers of safety-critical computer systems have to deal with multiple different topics:

Embedded systems are rapidly growing in complexity: With the decrease of hardware costs more multi purpose processors are employed and customized by software implementation for specific applications. Additionally the increase of hardware performance leads to a trend of integrating multiple functions into single electronic control units (ECU) for cost reductions (*e. g.*, in automotive and aerospace domain). This approach makes high demands to software middleware for keeping application programming manageable and nevertheless result in safe systems.

Real-time systems have special demands on the response time behavior. The claim to control a dynamic physical system requires reactions at special points in time and not only correctly calculated values. Operating systems and application programs need to consider timing behavior in order to achieve determinism in the temporal domain.

Distributed systems are employed to cope with redundancy and wiring complexity. Introduction of "brainpower" for so-far simple system entities as for example smart-sensors and new strategies for safety driven redundancy employment increases communication needs between ECUs. Shared communication bus systems replace point to point connections. For different applications and safety requirements, different methods for shared media access control have been employed. For "normal"-criticality applications event-driven communication like Ethernet or CAN have long been established. Safety-critical applications demand deterministic fault-tolerant communication schemes and therefore sophisticated communication protocols, for example based on the *time-triggered* paradigm like the *Time-Triggered Protocol* (TTP) [1], [2] enable the use of bus-communication with COTS products.

To cope with the growth of complexity in all of these areas, *model-based* system design is proposed for development. Graphical representations augment or replace textual implementations for illustrating and documenting parts of a system. Models display information usually in a way optimized for human perception; nevertheless the information can be stored and processed with machine readable interfaces. Hence the content of high abstraction models can be used by appropriate tool-chains to further process the data and relieve the developer of work in lower abstraction levels. For example models can describe configuration of systems (*e. g.*, describing distribution of the system onto multiple ECUs and the communication behavior between them). Post-processing tool-chains use the model information to automatically configure the operating systems and communication controllers (*e. g.*, building communication schedules in time-triggered communication). Other model types describe the functional behavior of a system. Model suites such as *Matlab Simulink* [3] or *SCADE* of *Esterel Technologies* [4] allow to specify data flow of the application in a way well suited for closed control loop design. The *Statechart* [5] formalism and dialects like *Stateflow* [6] and the synchronous *SyncCharts* [7] (with minor modifications known as *Safe State Machines* (*SSM*) [8]) are used to model control flow of reactive applications. Usually models inherit such precise semantics that they can be executed. On the one hand simulation is used to validate system behavior in a very early stage of the development process, especially prior to physical integration, and on the other hand generators can synthesize code for the target platform in order to directly derive the target implementation from models.

An international research project tries to focus not only on one of the issues, but tries to improve and combine them in order to establish a seamless process and a whole architecture for the development of embedded systems: The DECOS project (*Dependable Embedded Components and Systems*) [9], an Integrated Project within the EU Framework Programme 6, develops fundamental and enabling technologies that are independent of domain and technology in order to facilitate the paradigm shift from *federated* to *integrated* design of dependable real-time embedded systems. This shall lead to reduced development, validation and maintenance costs

(a) Top View.

(b) Bottom View.

Fig. 1: Overview of the Model Railway.

in both software and hardware domain. The project tackles all aforementioned topics, especially integration of multiple functions of different criticality into single ECUs and multiple virtual communication links into single physical communication channels based on the time-triggered communication approach. Additionally model based system design is used to specify configuration (in UML models) and functionality (in SCADE models) and a tool-chain for a seamless process from requirements capturing upto code generation and software deployment is developed.

As outlined above, a developer of embedded systems has to consider many topics and therefore needs a certain amount of know-how to participate or manage application development. The quality of the development results highly depends on the qualification of the development team even with massive tool support. This fact accentuates the need for a specialized qualification. The group of embedded and real-time systems of the Christian-Albrechts Universität Kiel offers a balanced combination by providing both theoretic knowledge and hands-on seminars. Imparting an extensive understanding of fundamental paradigms with a detailed theoretical background is the primary objective of university education. Advanced laboratory courses are employed to deepen the subject and accumulate some practical experience. The largest demonstrator currently in use by the embedded systems group is a model railway installation that originated about ten years ago and was used intensely since then [10], [11], [12]. Its controlling electronics has recently been replaced by a framework supporting multiple distributed architectures and bus systems [13]. This results in a multi-purpose demonstrator for different platforms, different bus systems and different programming paradigms, including model based system design, so the system tackles many of the topics of the embedded system domain.

As a part of the DECOS training activities, this demonstrator was used in an advanced laboratory course. The distributed application was requested to be implemented by the students using a time-triggered architecture at a high abstraction level employing model-based system design.

II. The Model Railway

The model railway[1] was inspired by the Kicking Horse Pass in Canada, from which it also derives its name. Currently, the layout covers an area of 18 square meters, in which up to eight trains can pass along three interconnected major circles (inner circle, outer circle and the Kicking Horse Pass itself) that form a total track length of 127 meters. Figure 1 provides a general view of the model railway. The schematic track layout is given in Figure 2.

The railway hardware is a composition of relatively dumb analog peripherals operated by custom made controller boards under the supervision of networked computers. Using a digital system was not an option. This would have eliminated the need for distributed communication and would thus have made the system pointless for embedded system education.

Any applications that intend to move trains have to utilize the communication networks in order to address the controller boards with their attached sensors and actuators. The most prominent actuators are 48 isolated track sections (or blocks) to which traction current for the train engines can be applied. A task central to all of the control programs is that of keeping track of the train positions and switching power for the individual blocks. This is designed to ensure seamless transitions of the trains through the various blocks. A group of 80 reed contact sensors, which detect trains as they enter and leave individual blocks, feeds basic information into this process (Figure 3a). The lane selection is done by 28 switching points; 58 semaphores and 24 lanterns (Figure 3c) can be used to mimic realistic railway behavior and to visualize the current system state. Including devices like the railroad

[1]The hardware technical basics for the model railway project emanates from the diploma thesis of Höhrmann [13]. Further details concerning the model railways technical realization as well its functionality can be found on the model railway web page [14].

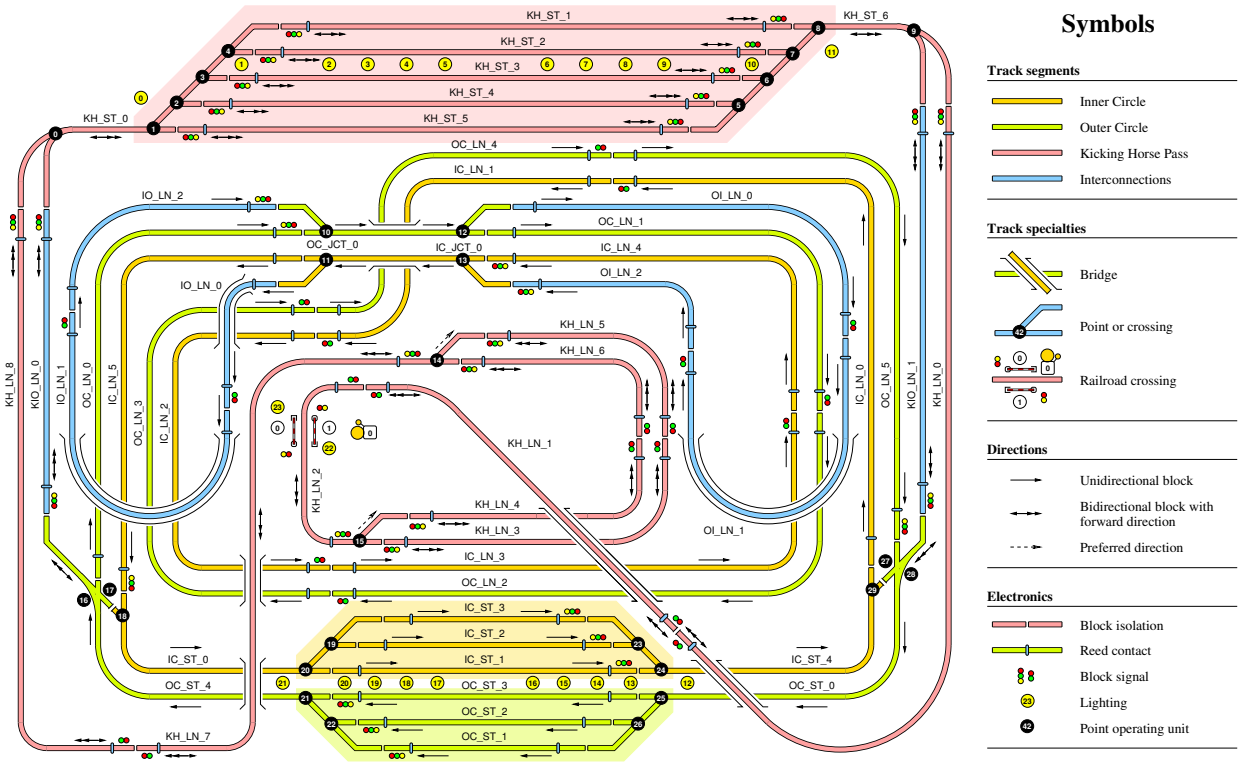
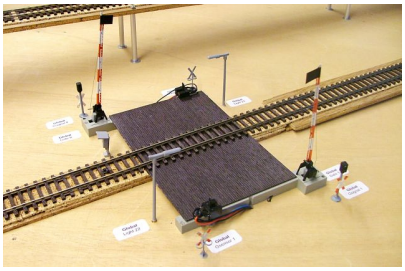


Fig. 2: The Track Layout. The picture illustrates the track organization including all block names, sensors and actuators.

(a) Reed Contact Sensors.



(b) Railroad Crossing.

(c) A Railway Station with Switches, Semaphores, and Lanterns.

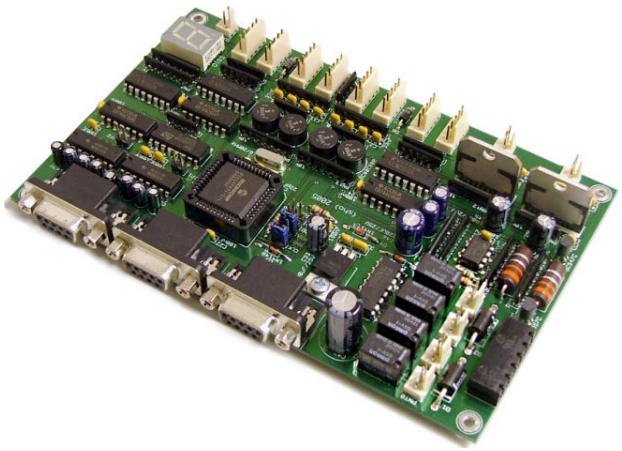
Fig. 3: Railway Sensors and Actuators.

crossing (Figure 3b), a total of 245 sensors and actuators can individually be addressed and programmed. All peripherals are wired to a total of 24 hardware controller nodes (see Figure 4a). These embedded devices are operated by a PIC 16F871 microcontroller and comprise all features required to drive the connected peripherals.

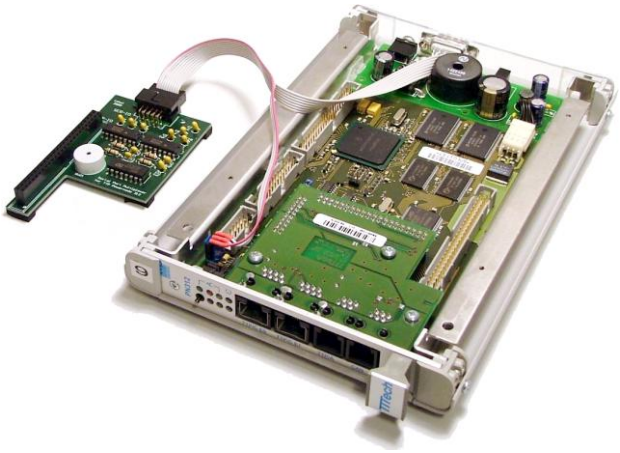
As well as basic actions, like enabling semaphore LEDs and toggling switching points, the microcontroller executes several advanced control tasks. Back electromagnetic field (EMF) measurement on the track power supply lines is used to detect occupied blocks, the corresponding circuit is able to acquire the actual speed of driving engines. This information can be used in a closed control loop to adjust the output Pulse-width modulation (PWM) duty cycle to maintain a given speed *e. g.*, in case of a train traversing an incline. Sensor readings from the reed contacts are processed to determine the direction in which the train has passed. Hardware malfunctions like short circuits or sensor failures are logged in non-volatile memory for later examination.

The full functionality of the board is available through RS232 compatible control inputs (cf. Figure 5a). Up to four computers can connect mutually exclusively to a node by executing a simple request/reply protocol. The microcontroller attaches itself to the first input where it observes valid communication activities. It stays locked until either the host sends a termination command or stops sending valid packets. This enables the system to rotate through different hosts without modifications to the hardware.

Following the initial setup procedure, control commands can be sent to operate peripherals and to query the attached

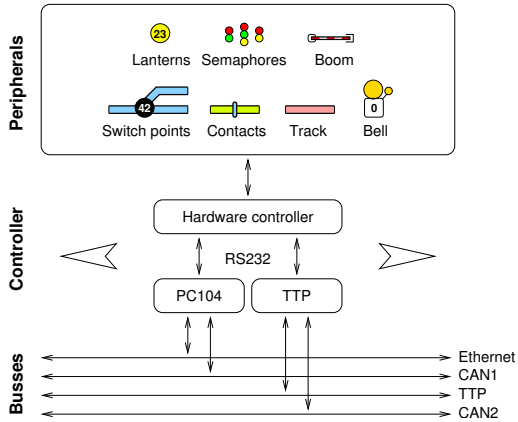


(a) PC Node.

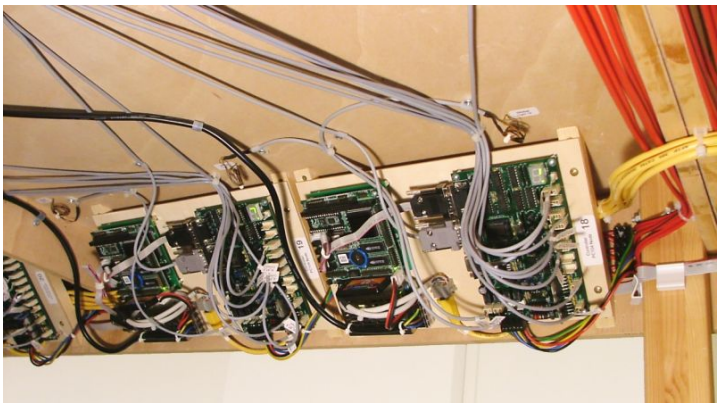


(b) TTP-Powernode with Portextender.

Fig. 4: Hardware Control Nodes.



(a) Schematic Representation.



(b) Wiring of Control Nodes.

Fig. 5: Control System Interconnections.

sensors. This is the task of the computers installed in the system. The first cluster consists of 24 PCs using the PC104 form factor. These machines boot Linux via network on a dedicated server, thus they are implicitly connected via Ethernet. Additional CAN controllers provide an independent communication layer. Each of these computers is mounted on a wooden panel inside the railway under-structure together with its power supply and a hardware controller board (see Figure. 5b). The PC104 hosts can execute cross-compiled programs previously copied to the server.

The second control cluster consists of eight TTP-Powernodes. A custom built extension board (see Figure. 4b) multiplexes all of the node's single RS232 ports, allowing each one to handle three hardware controllers. The TTP-nodes rely mainly on TTP communication. Additionally they are equipped with an independent CAN bus interface.

The two remaining inputs of the hardware controller nodes are available for future extensions, but the two already existing control systems and four independent networks offer a wide range of applications. Several design strategies can be realized and evaluated in this complex environment.

III. Model Driven Programming

Implementing application software for embedded systems in C or C++ is quite common, so appropriate compilers exist for most processors. Next to textual programming with the imperative languages themselves, any system design paradigm is suitable for development that produces standard C code. Actual modeling suites for functional models, *e. g.*, in dataflow style or a Statechart derivate, usually provide C code generators. For one architecture of the model railway this has been utilized: Dataflow models and Statecharts were used to model functional behavior of individual ECUs, namely controllers equipped with a Motorola Power PC processor and a TTP communication interface.

In order to keep the abstraction level as high as possible, manual integration of the functional software parts onto the distributed ECUs can be avoided: Distribution and configu-

ration of the communication behavior of the whole system is modeled, too. This includes a mapping of functional parts modeled in data flow models/Statecharts to certain hardware units. A tool-chain automatically generates the needed middleware and operating system configurations for each node, merges configuration and functional code, invokes the compiler and finally deploys the binaries to the nodes through the bus system automatically.

Some different modeling suites and tool-chains would be applicable, but since neither the DECOS tool-chain nor the SCADE specific tool-chain *SCADElink* were available at the beginning of the planning, Matlab Simulink and Stateflow were employed. The Simulink add-on *Matlink* from *TTTech* was used to configure the distribution and time-triggered system behavior directly in the familiar surroundings of Simulink.

IV. Practical Use in Education

In the winter term 2005/2006 an advanced laboratory course was offered that employed the model railway demonstrator for the first time in teaching after finalization of the hardware [15]. According to the goals of the DECOS project, the main focus of the course was *model-based system design of a time-triggered architecture*. For this first approach one architecture and one tool-chain were employed in order to gain experience with the demonstrator in education. The platform chosen for the implementation was the *Time-Triggered Architecture* (TTA) with the Matlab/Simulink tool-chain introduced above.

Imparting knowledge usually requires finding a tradeoff between specific technical details and deeper insight about the general paradigms. The former is necessary to accomplish the specific task, but the latter is more fruitful for the long term. Using the modeling suite with its high abstraction level covers technical details of both model railway hardware drivers and the time-triggered protocol.

Hardware drivers that communicate directly with the hardware controller nodes were prepared beforehand the course, they can be accessed conveniently via Simulink blocks directly from within Simulink.

A. Background

The laboratory was attended by eight students that formed four teams. Most of them previously attended the class *model-based system design and distributed real-time systems* [16]. This class established a foundation for model-based system design with a special focus on Statecharts and their semantics as well as on distributed systems. With the time-triggered architecture being only one part, the basic ideas and algorithms were presented, but technical insights and practical experience with any tool-chain could not be expected.

Simulink as one of several possible tools was not yet explicitly addressed and therefore the model railway course was the first occasion where the students could gather experience with the technical pitfalls of the general purpose suite Simulink. Excessive experience in C programming could not be expected, because other programming languages (*e. g.*, Java) are often preferred in education.

B. Requirements and Process

The course started with independent one-week exercises:

- A small Simulink model had to be analyzed and adapted in order to get familiar with the Simulink development environment.
- A toy example (LED blinking) of time triggered communication had to be modeled in Simulink/Matlink and tested on the hardware in order to get familiar with the Matlink tool.
- On a small testing track oval, one ECU was connected to a few model railway peripherals in order to access the hardware drivers and the railway itself. A trivial controller had to be implemented using Stateflow. Additionally a closed-loop controller had to be realized in Simulink to keep the train speed at a defined level. Knowledge about PID-controllers had to be acquired autonomously.

One major aim of this course was to work on a non-toy-size project while acting on one's own initiative. Therefore, after these tutorials, the general task for all students was to control the model railway in some way. All groups had to specify an application for the model railway that demonstrates the model-based system design and fits to their individual timeframe (some students attended four hours, others eight hours a week). Explicit interaction among the groups was allowed in order to get experience in inter-group communication and in defining interfaces and standards within all groups.

The time until end-of-term was divided to four intermediate and a final milestone. Each group had to do the time planning itself, including to define sub-ordinate goals for each milestone for their application. A full documentation of the results was requested to the end of the course, whereas an explicit reviewing process between the groups should improve documentation styles. At each milestone a short presentation should demonstrate the achieved results so far and a final presentation was used to present the final work to a larger audience. The presentations and documentations are available as a project report [15].

C. Technical Results

1) Simulation: One aim of model-based system design is to find design flaws in an early design phase. Therefore many modeling suites offer the possibility to execute the model prior to testing the code on physical hardware. Controllers usually interact with actuators and sensors to form any kind of control loops. Therefore in such a dry-run some appropriate sensor values must be provided for the controller and its actuator commands need to be consumed. Therfore an environment simulation must be implemented. In our case another Simulink model represents the physical railway hardware itself. It takes commands from a controller containing the whole actuator state, *e. g.*, speed commands for each railway block, track switch positions, semaphores, LEDs, lanterns, *etc.*. The simulation should behave in the same manner as the real model railway does and return corresponding sensor values corresponding to state changes at the railway tracks. The

simulation primarily needs to keep track of the train positions and trigger contact passing events and pass them back to the controller. With a simulation model being application compatible, the test of the application is easily possible prior to testing on the physical installation. Debugging the controller is greatly simplified this way, because the internal states of the controllers can always be observed and test cases can be run much faster (returning physical trains to the initial positions is unnecessary) and faults cannot cause any harm.

Because of all these advantages and the high complexity of this model railway installation, the development of a controller required a simulation model. One student group planned to build such a simulation model for the whole track layout and some graphical user interface to present the railway behavior to the developer. The model should be modular and as general as possible.

2) Controllers: All other groups decided to implement railway controllers (*e. g.*, see Figure 6) for different purposes, varying in the number of concurrently controlled trains and the degree of independence of the physical track layout. One group implemented a switching application in which one engine independently searches the shortest path to a trailer and pulls it to some destination place. Another group planned to control up to four trains concurrently and were exposed to requirements of collision and deadlock avoidance as well as fairness. The controller groups were supposed to employ the simulation of the first group for testing. However development of simulation and controllers was happening in parallel and therefore the controller groups could not access the simulation from the beginning. As simulation was a mandatory requirement for testing before hardware integration, the controller groups had to implement their own simulations. Nevertheless these were allowed to be very application specific and simplified as much as possible.

3) Results: Prior to actual controller logic implementation, the teams had to model the communication between the distributed hardware nodes. Although this was already exercised in a toy example, it became a laborious task for the whole system. 744 signals from sensors and actuators needed to be processed, routed and sorted. For wiring simplicity, the physical sensors and actuators became connected to the spatially nearest hardware control nodes not considering any logical order. Hence one ECU could be responsible for two railway tracks that have little relations to each other, *e. g.*, that are not adjacent. Therefore a sorting subsystem, the so-called *dispatcher*, was built to access sensors and actuators in simulation and controller in a somewhat convenient way. Multiple signals became grouped to arrays of up to about hundred elements to cope with the mass of information. Building up a schedulable communication model for the time-triggered communication lead to many technical and time consuming pitfalls.

The simulation model came out to be even more complex than the controller models and therefore all groups spent most of the time to implement their simulation models. This includes the controller groups, because nearly all models

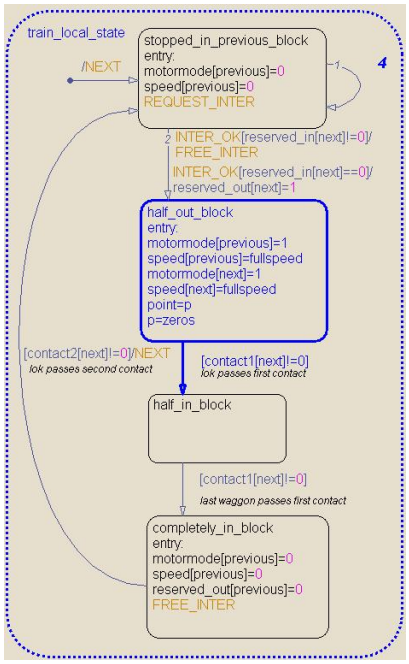


Fig. 6: Small part of a controller Stateflow chart keeping track of the position of one train in one railway block.

required a general simulation model that hardly could be simplified.

In the end the simulation group came up with a very well modularized simulation model, consisting of parameterizable building bricks for tracks and track switches. These could be employed in order to easily and clearly build up any track layout. The first connected (but only working for uni-directional train movement) simulation was available at the end of the course. This meant that the other groups had to use their own simulations, that were less modular with usually very complex and track layout specific connections.

V. Lessons Learned

A. Motivation

The motivation during most of the time of the project was high. The presence of a physical demonstrator seemed to positively motivate the students. Hence most of them spent much more time at the subject than scheduled. Practical exercise not only on abstract modeling examples but with intermediate testing of the subordinate results at the physical installation was positively acknowledged. Nevertheless preparations before even starting to develop a controller logic were very laborious (modeling simulation, communication model, dispatcher, *etc.*), but having the target of an autonomously controlled model railway in mind, these tasks were finished with interest.

B. Time Planning

It seems to be basic constitution in computer science, that time planning is always too optimistic. All groups underestimated the efforts of their tasks and were hardly able to finish

them in time. Most groups voluntarily invested much more time than scheduled (*e. g.*, more than sixteen hours a week instead of eight) and nevertheless some of the goals for the milestone were postponed or even removed from the plans.

C. Preparation

Most problems derive from technical difficulties with the tool suites. Missing experience with Simulink resulted in error-prone and time consuming trial-and-error approaches or searching the Matlab manuals into wrong directions. The same was true for Matlink during specification of the time-triggered communication. Generation of communication and task schedules from the models often resulted in error messages that were of very technical nature, and finding the cause and especially a way to fix it was a very effort-prone task.

D. Task Scale

Implementing either a simulation or a controller for the model railway seems to be manageable within the given time-frame. However doing both successively obviously overstrained the groups. Unfortunately the two tasks could hardly be parallelized: Development of the controller requires a nearly full featured simulation.

E. Application

Building a simulation model of the model railway fitted very well to the dataflow model paradigm of Simulink, because it mostly consists of signal routing and keeping track of positions by aggregation of train speeds. The controller, however, shows heterogeneous characteristics: Signal pre- and post-processing (*e. g.*, *dispatcher*) is suitable for the dataflow style like used in Simulink. Managing train movements in respect to collision and deadlock avoidance and fairness can be covered in control flow models like Statecharts. An additional task for a controller can be to autonomously find a path from one point of the track layout to another one. Such task needs to implement some kind of graph algorithm for shortest path findings. Such algorithms neither apply to dataflow nor control flow paradigms and therefore they need to be implemented in a host language, *e. g.*, the Matlab language or C for Simulink. Figure 7 presents concurrently running trains operated by a control application.

F. Simulation

Although implementing a simulation model was very laborious, it was worth the effort: After extensive testing of the controller with the simulation model, migration to the physical railway installation could be done smoothly and the controllers worked with nearly no further debugging on the real hardware. The other way round, at the end of the course one group tried to fix some buggy behavior by testing the controller directly with the physical railway in order to avoid the connection to the simulation model. They spent many hours to find the bug, because the internal states of the Statecharts could hardly been analyzed and the behavior was not comprehended.

Fig. 7: Concurrently running trains operated by a control application.

G. Experience at Modeling

Models are easy to read but hard to write. This sentence is very true especially for beginners. Modeling patterns usually differ from programming patterns. Having only little knowledge about how to write good models ends up in a mess of wirings and transitions, where even the developer him- or herself could loose the overview. By using a quick-and-dirty approach it is easy to reach a functional state where even simulation is possible. Impatient students usually do not take time to rearrange the layout for a better view onto the model or to group multiple signals into arrays or structures in order to massively reduce the number of visible connections and interfaces between model elements. States of control flow models get abused to reflect data flow instead of representing an actual system state. The decision of what functionality requires to what kind of model seems to be very difficult. The right balance of applying hierarchy is often not found. Stateflow's inter-level-transitions make reading the models much harder.

After discovering the feature of embedding sequential textual code in the Matlab language in Simulink/Stateflow, some of the students used it excessively. A lot of functionality was implemented in Matlab functions in order to elude complicated graphical constructs in the modeling language itself. Usually these functionality could be clearly expressed in the modeling language too, but the students avoided the work of finding out the correct ways and sticked to the textual programming style they were familiar with.

The *write-things-once* demand is one of the most established paradigms in computer science. Nearly every programming language provides constructs for implementation of system parts that can be called from multiple sources, *e. g.*, functions or procedures. In the dataflow models of *SCADE*, this idea is consequently implemented, but in Simulink it is neglected. Once they are implemented, subsystems usually get multiplied by copy-and-paste, which results in laborious actions if any changes in the subsystem have to be done. Using libraries in

Simulink is not as obvious and established as it could be and sometimes leads to unexpected technical problems. Writing "beautiful" models is possible, but creating an unreadable chaos is much simpler. The latter kinds are even produced by experienced engineers. Writing good models is less a matter of experience than a matter of the right examples.

H. Interaction with the Tools

Integrated tool-chains help the developers to focus on the relevant design decisions. Functional modeling, configuration modeling, schedule generation, middleware generation, functional code generation, code merging, integration and compilation: each of these tasks could be done by a certain tool, not necessarily provided by the same vendor. In a perfect process, the developer stays at the highest level of abstraction. Obviously changes in a model affect many of the lower levels, but the idea is that the developer is not bothered with these details.

Problems arise when changes to the model, that are not rejected by the modeling tool, result in an error message from some lower level program. Exactly this has happened repeatedly during the project and was quite irritating and time consuming for the novice tool users. This issue can be observed with many tool-chains and should explicitly be tackled in order to increase usability to such extend that professional help for debugging is not necessarily needed. This should also increase acceptance and motivation in the use of the tool-chains.

VI. Conclusion und Future Work

We have proposed a model railway demonstrator that tackles many topics of the embedded systems domain and supports multiple paradigms such as different architectures, bus-systems and software development paradigms.

An advanced laboratory course was performed with this demonstrator as part of the DECOS training activities. Students implemented simulations and controllers for autonomous operation of the model railway. Model-based system design for a time-triggered architecture was employed and lead within the time-frame to acceptable results.

The physical demonstrator increased the motivation of the students, while underestimation of the effort and unexpected problems of technical nature were challenging and nearly doubled the expenditure of time. For a demonstrator of this scale with hundreds inputs and outputs to the physical environment, running tests with a simulation model is not only helpful, but mandatory for debugging and validation.

Although graphical modeling looks simple, basic knowledge of computer science is not enough to generate good and reusable models, and only such expose the benefit of model-based system design. As this project has confirmed, the proper use of complex tool-chains and advanced system architectures also requires specific education and training in these areas. This has to include not only the theoretical background, but also engineering know-how on how to construct robust, maintainable models and, last but not least, how to take advantage of team-oriented development processes.

Acknowledgment

The authors would like to thank Werner Kluge, Jürgen Noss, Jochen Koberstein and Oliver Schmitz for the first generations of the model railway demonstrator.

References

[1] H. Kopetz and G. Grünsteidl, "TTP - a time-triggered protocol for fault-tolerant real-time systems," Institut für Technische Informatik, Technische Universität Wien, Treilstr. 3/182/1, A-1040 Vienna, Austria, Tech. Rep., 1992.

[2] H. Kopetz and G. Bauer, "The time-triggered architecture." *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[3] Mathworks Inc., *Simulink – Simulation and Model-Based Design*, Mathworks Inc., 2005. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sl_using.pdf

[4] Esterel Technologies, "Company homepage," http://www.esterel-technologies.com.

[5] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[6] T. MathWorks, Inc., *Stateflow and Stateflow Coder User's Guide, Version 6*, Natick, MA, September 2005. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/pdf_doc/stateflow/sf_ug.pdf

[7] C. André, "SyncCharts: A Visual Representation of Reactive Behaviors," I3S, Sophia-Antipolis, France, Tech. Rep. RR 95–52, rev. RR (96–56), Rev. April 1996. [Online]. Available: http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf

[8] C. André, "Semantics of S.S.M (Safe State Machine)," Esterel Technologies, Sophia-Antipolis, France, Tech. Rep., Apr. 2003, available at http://www.esterel-technologies.com, in the download section.

[9] DECOS - Dependable Components and Systems, "Research project homepage," https://www.decos.at/.

[10] W. Hielscher, L. Urbszat, C. Reinke, and W. Kluge, "On modelling train traffic in a model train system," in *Daimi PB-532: Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, K. Jensen, Ed. Department of Computer Science, University of Aarhus, Denmark, 1998, pp. 83–102.

[11] W. E. Kluge, "The Kicking Horse Pass Problem," *Petri Net News Letters No. 54*, pp. 3–15, 1998.

[12] J. Koberstein, "Realisierung eines geordneten Mehrzugbetriebs auf einer Modellbahnanlage," Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und praktische Mathematik, 2001.

[13] S. Höhrmann, "Entwicklung eines modularen Feldbussystems zur Steuerung einer Modellbahnanlage," Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik, Mar. 2006.

[14] The model railway, "Project homepage," 2006, Group of Real-Time and Embedded Systems, Department of Computer Science, Kiel, Germany. [Online]. Available: http://www.informatik.uni-kiel.de/~railway

[15] H. Fuhrmann, S. Prochnow, and R. von Hanxleden, "Modellbahnpraktikum," http://www.informatik.uni-kiel.de/inf/von-Hanxleden/teaching/ws05-06/p-bahn/, 2005/2006.

[16] R. von Hanxleden, "Lectures: Model-based design and distributed real-time systems," http://www.informatik.uni-kiel.de/inf/von-Hanxleden/teaching/ss05/v-rt2/skript.html, 2005.