Hierarchical Simulation Techniques for the Design of Heterogeneous, Embedded Systems

Ulrich Freund, Gunar Schorcht, Volker Zerbe
Department of Computer Science and Automatic Control
Technical University of Ilmenau
Postbox 100565
98684 Ilmenau,Germany
Phone ++49/3677/69-1201
Fax ++49/3677/69-1285
{freund,schorcht,zerbe}@theoinf.tu-ilmenau.de

October 15, 1997

Abstract

In this paper we present the HiFAQ Modelling and Simulation language. This language allows the assessment of the functional and performance properties of a heterogeneous embedded system by means of integrated, timed simulation. A typical fly-by-wire control system is modelled by means of the HiFAQ technique showing the effects of different implementations on the dynamics of the control loop
Keywords: *Integrated System Level Simulation, Embedded Systems, Heterogeneous Systems*

1 Introduction

Currently, designers of embedded systems are still forced to determine the architecture at an early stage in the design cycle. Examples are Hardware/Software partitioning, type and number of microprocessors and microcontrollers, scheduling strategies, and communication protocols. A full redesign is necessary if the chosen architecture is not capable of performing the function within the performance requirements. On the other hand, it might happen that the resources of the architecture are not used completely which might result in a too expensive product.

These limitations are well recognised in the aerospace [1] and automotive industries [9, 6, 3, 5, 4] and could be overcome by means of the model based design approach . However, the coupling of two or three tools is still necessary to asses the effects of an architecture to the system dynamics. To overcome this situation we are introducing the HiFAQ modelling and simulation language.

The paper is organised as follows. After reviewing the model based system design approach we introduce the formalisms behind the HiFAQ's. In section 3, the language is described and the class library necessary for combined simulation is introduced in section 4. Section 5 shows how the architecture can influence the dynamics of a control loop.

2 Model Based System Design

Using the model based system design approach, the embedded systems model is divided into an architectural and a functional model. Functional models consist of transformatorial models, reactive models or a combination of both. Architectural models for the early design phase are transaction/resource models, which can assess the performance properties of the system in the early design phase. These three different layers are shown in figure 1 The reason for putting the continuous layer on the top is that

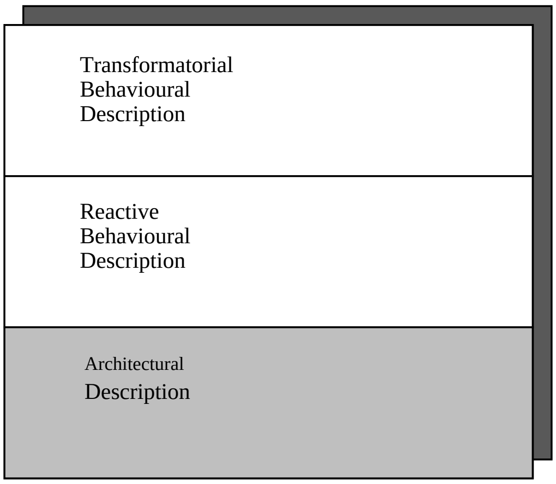

Figure 1: The different levels of a behavioural- and performance-model.

we are dealing with digital systems. This includes sampled continuous systems like digital controllers for continuous systems as well as algorithms for digital transmission systems. Remember, the target of design is the implementation of Real-Time Systems in HW/SW.

The functional model can be mapped onto several architectural models. Architectural models represent alternative HW/SW parts. Using this approach, the designer gets an immediate feedback by means of an integrated timed simulation run to which extend the architecture will fit the function. Unsuitable architectures are ruled out instantly. This is depicted in Fig. 2

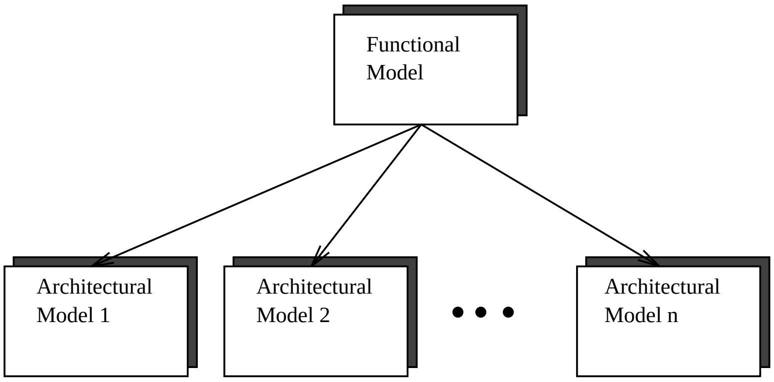

Figure 2: Mapping a functional description to an architecture

3 The HiFAQ Approach

The above approach requires a tool capable of modelling and simulating heterogeneous, embedded systems. In this paper we present a modelling and simulation language for the design of those systems. The main elements of this language are the HiFAQ's (Hierarchical Functional Augmented Queuing Networks). HiFAQ's consist of three layers: the transformatorial, the reactive and the architectural layer. Elements of the HiFAQ's are functional and architectural model elements. Each element is associated to one of the three layers. After the interfaces between the layers are defined, the layers can be modelled separately. Depending on the point of view, one gets a transformatorial functional view, a reactive functional view or an architectural/resource oriented view. Through mapping of the functional models onto the architectural models the effects of architecture modifications are unveiled via simulation.

Within a HiFAQ the reactive elements are represented by extended finite state machines, and the transformatorial layer's elements are described via difference/differential equations. The architecture is modelled by transaction/resource models. The coupling between the layers of a HiFAQ is realised via a parameter state coupling. Within a layer the coupling is realised by means of input/output coupling. HiFAQ's themselves are coupled via their inputs and outputs. These couplings can carry transformatorial, reactive and architectural information.

Hardware parts are modelled by transaction/resource models dedicated to a functional model exclusively. A similar approach is used for software parts. Every software process is assigned to a transaction. The processor is modelled as a resource. The processor time required by a transaction is determined by the transition times of the finite state machine representing the software process. However, in a multitasking single processor system, the process' transactions are waiting in a queue for the processor, i.e. sharing the resource.

A system model can be described by a matrix. Each layer constitutes a row of similar models. The model can therefor viewed from the transformatorial layer, the reactive layer or the architectural layer. The columns distinguish between synchronous or asynchronous communication, representing the HiFAQ's. Figure 3 shows this matrix.

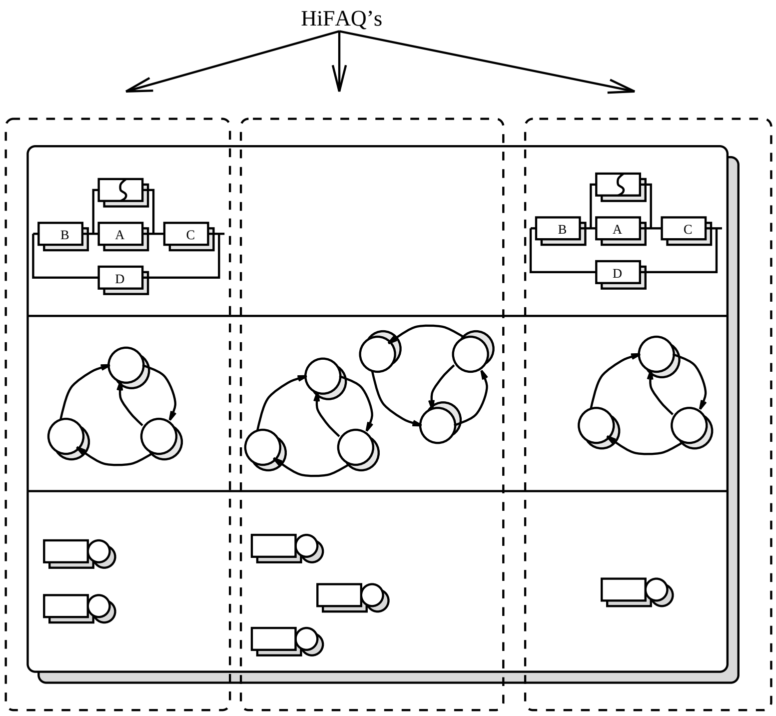

Figure 3: Modelmatrix

3.1 System Theoretic Foundations

The main idea of HiFAQ's is to develop a combined functional model. From this model, one extracts the timing and resource information and puts this to the architectural layer. In the following, we show which descriptions are used in HiFAQ's and how they are coupled to obtain an integrated model.

3.2 Preliminaries

- $||$ is the concatenation operator
- $\mid S \mid$ denotes the number of elements in set S
- * is the Kleene operator on strings
- ϵ is the null string
- *Time(e)* is the event-time of event e
- $Proc_{Time}(T)$ is the processing duration for transition T
- P_f is a multiset [1]

3.3 continuous systems

A continuous dynamical system is a five-tupel [U,X,Y,f,g] where

- U $\in R^n$ is the input vector,
- X $\in R^m$ is the state vector,
- Y $\in R^k$ is the output vector,
- f:U$\times$X $\longrightarrow$ X is the state transistion funcition
- g:U$\times$X $\longrightarrow$ Y is the output function

f and g describe the dynamics of the system. f is a system of ordinary differential equations.

$$\dot{x} = f(x, u)$$

$$y = g(x, u)$$

This is the description of a nonlinear system in normal form.

3.4 EFSM's

An extended finite state machine (EFSM) extends an ordinary finite state machine by means of Variables. Variables can be of real or integer type. An EFSM is defined by a five-tuple [X,Z,Y,P,δ,λ] for the Mealy type and [X,Z,Y,Pδ,μ] for the Moore type. For the Mealy type we have:

- X is the input alphabet
- Z is the state set

[1] a multiset of a set M is a set where the elements of M may be multiply included. In contrast to a string, there is no ordering defined.

- Y is the output alphabet
- P is the Variable set with p $\in \mathcal{P}$ or $\mathcal{R}$
- δ : X $\times$ Z $\times$ P $\longrightarrow$ Z $\times$ P is the state transition where also parameters may be changed.
- λ : X $\times$ Z $\times$ P $\longrightarrow$ Y is the output function

An extended finite state machine is used to describe communcitation protocols. One popular representive of this class is SDL.

3.5 GSMS

A generalised semi markov scheme (GSMS) can represent a timed-automaton (TA) described by a five-tuple [12] $TA = [E, H, \Gamma(H), f, G]$ for the case that we only consider deterministic state transitions.

- E is a countable event set
- H is a countable state set
- f is the state transition function with,

$$f(H, E) \rightarrow H$$

- and,
- $\Gamma(H)$ is the function of feasible events in state h $\in H$.
- G is a set of pdf's of interarrival times for each event.

In some sense, a FSM is a special case of a GSMS with the main difference that the state-space of a GSMS is countable infinite whereas the state-space of a FSM is finite. This difference is necessary e.g. to model queuing systems.

TA represents an automaton that is driven by a timed input stream

$$EL : \{e_1, e_2, ...\} = P_f(E)$$

Every element of the input stream has an associated time-stamp. Hence, considering only time, we have a time-sequence

$$T_s : \{t_{1,e_1}, t_{2,e_2}, ...\} = \{Time_{e_1}, Time_{e_2}, ...\}$$

Such a system can only change its state at time-points t_i. The state-change is given by

$$h_{new} = f(h_{old}, \tilde{e})$$

with

$$\begin{aligned} \tilde{e} &= min_{i \in \Gamma(h)}(t_i) && (1) \\ &= min_{i \in \Gamma(h)}(Time_{e_i}) && (2) \\ &= argmin_{i \in \Gamma(h)}(e_i) && (3) \end{aligned}$$

During the state-transition new events may become feasible while others become infeasible. To describe this, we introduce the set of the new feasible events with $\pi(E) \subseteq P_f(E)$ and the set of infeasible events $\tau(E) \subseteq P_f(E)$. The input-stream or event-list (EL) of a TA changes during a transition $f(h, \tilde{e})$:

$$\Gamma(h_{new}) : P_f(EL_{new}) = (P_f(EL_{old}) \setminus \tilde{e}) \setminus \tau(E) \cup \pi(E) \tag{4}$$

Equation 4 describes the behaviour of the eventlist.

3.6 From GSMS to Queueing-Systems

3.6.1 Queueing Systems

A queuing-system (QS) is a GSMS with $[E, H, f, \Gamma(h), G]$.

- E is the eventset with {a,d} where a denotes the arrival event and d denotes the departure event
- H is state-space with {1,2,...} and denotes the number of customers in the queue.

$$E = \{a, d\}$$

$$\Gamma(h) = \{a, d\} \forall_{h \geq 0}$$

$$\Gamma(0) = \{a\}$$

$$f(h, \tilde{e}) = \begin{cases} h+1 & \text{if } \tilde{e} = a \\ h-1 & \text{if } \tilde{e} = d \text{ and } h \geq 0 \end{cases}$$

$$G : pdfs$$

The state of the queuing system is the queue-length. The event-list consists of arrival and departure events. The time between two arrival events is called interarrival time and is denoted with λ. The time between the two consequetive departure events is called service time and denoted with μ. This might lead to some confusion with a Moore's FSA output functions. λ and μ are probabilistic values of arbitrary pdfs $g \in G$.

3.7 Expanded Queuing Systems

Queuing systems are an abstract representation of Queues with servers. For analysis, the identity of the events is lost. However, if we want to map FSM's or EFSM's to architectural models, we need to keep the identity of every incoming event. Therefore, we have to track the sequence of the events in the queue. This requirement leads to the Expanded Queuing Systems (EQS):

The state of an EQS is defined as follows:

$EQS = [E_e, H_e, f_e, \Gamma_e(H), G_e]$ The eventset E is defined by

$$E = E_a \cup E_d \tag{5}$$

The output signals are the released customers of the QS and therefore equivalent to the arrival eventset E_a. The state set H can be described as sequence:

$$H = [h_1, h_2, ...] = \{E_a\}^* \tag{6}$$

h_i defines the the place of the event in the queue.

Like the eventset, we can divide the statetransition of the Queuing System by arrival transitions and by departure transition. In the case of an arrival event we have

$$f_a(H, E_a) = \begin{cases} H || E_a & \text{if } \mid H \mid > 0 \\ h_1 & \text{if } \mid H \mid = 0 \end{cases} \tag{7}$$

In the case of a departure event we have to take the queuing strategy into account. In case of FIFO, the first element of the queue goes into service and, if the queue is not empty, all other elements go ahead one place in the queue.

$$f_d(H, E_d) = \begin{cases} H \setminus h_1 \wedge \forall_{i=2}^{n} h_{i-1} = h_i & \text{if } \mid H \mid > 1 \\ \epsilon & \text{if } \mid H \mid \leq 1 \end{cases} \tag{8}$$

The LIFO strategy looks like this:

$$f_d(H, E_d) = \begin{cases} H \setminus h_n & \text{if } \mid H \mid > 1 \\ \epsilon & \text{if } \mid H \mid \leq 1 \end{cases} \tag{9}$$

where n is the queue-length and h_n is the last element of the queue. h_i denotes the event on the i-th. position in the queue. The latter transition functions also show that arrival events are queued whereas departure events release one element of the queue.

Hence, the complete state-transition function is:

$$f_e(H, E) = f_a(H, E_a) \cup f_d(H, E_d) \tag{10}$$

The outputfunction $\Gamma(H)$ depends only on the state of the EQS.

$$\Gamma_e(h) = \{E\} \forall_{h \geq 0} \tag{11}$$

$$\Gamma_e(0) = \{E_a\} \tag{12}$$

The system time is the sum of the waiting time and the service time. The waiting time depends on the queuing-network's architecture.

We can define the following mapping of an EQS to a QS. The arrival eventset of the EQS can be mapped to the generic arrival event

$$A : E_a \longrightarrow a \tag{13}$$

The same holds for the departure event.

$$D : E_d \longrightarrow d \tag{14}$$

Concerning the state, dropping the identity of the events in the queue, one can map the state of a queue to its lengths:

$$S : H_e \longrightarrow \mathcal{N} \tag{15}$$

$$F : E_e \longrightarrow E \tag{16}$$

3.8 Coupling (transformatorial-/reactive functional)

To bring a continuous dynamik system to a digital implementation, one can identify three different sampling modes:

1. State-Event based sampling
2. equidistant sampling without delay
3. equidistant sampling with delay

3.8.1 The State-Event coupling:

For the combination of continuous and finite-state automaton let us consider the following things. The state space of the continuous system is explored by its transition function. However, there are barriers, which can be crossed by the state trajectory. When such a barrier is crossed, a state event can be generated. The sum of all barrierers in the continuous state space partions this into disjunct chunks. To each chunk, we can assign a discrete state. The same can be done with the input and output signal, with the difference that we assign events to the chunks.

We can express this situation formally by homomorphims. For the states spaces of continuous and finite state machine systems we have

$$h_1 : X \longrightarrow Z \tag{17}$$

The input is given by

$$h_2 : U \longrightarrow X \tag{18}$$

The output is given by

$$h_3 : Y \longrightarrow Y \tag{19}$$

For all three homomorphisms the following conditions have to hold:

$$\forall_{x,u} x \in X, u \in U : \delta(h_1(x), h_2(u)) = h_1(f(x,u)) \tag{20}$$
$$\forall_{x,u,y} x \in X, u \in U, y \in Y : \lambda(h_1(x), h_2(u) = h_3(g(x,u)) \tag{21}$$

This means that we have the same discrete and continuous states for the same situations, expressed in two different domains. This situation is depicted in Fig. 4

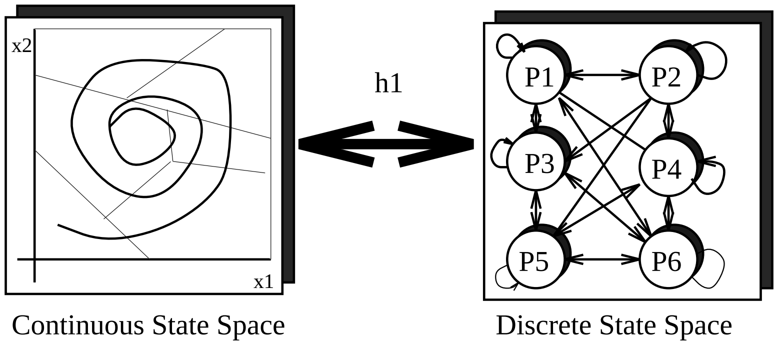

Figure 4: State-Event Mapping of continuous to discrete description

3.8.2 The equidistant sampling without delay:

We only consider the linear case, since relatively often a system is linearised around a setpoint.

$$x(kh + h) = \Phi x(kh) + \Gamma u(kh) \tag{22}$$
$$\Phi = e^{Ah} \tag{23}$$
$$y(kh) = Cx(kh) + Du(kh) \tag{24}$$
$$\Gamma = \int_0^h e^{As} ds B \tag{25}$$
$$t_k = kh \tag{26}$$
$$\tag{27}$$

3.8.3 The equidistant sampling in case of delays:

Like above, we here only consider the linear case.

$$x(kh+h) = \Phi x(kh) + \Gamma_0 u(kh) + \Gamma_1 u(kh-h) \tag{28}$$
$$\Phi = e^{Ah} \tag{29}$$
$$y(kh) = Cx(kh) + Du(kh) \tag{30}$$
$$\Gamma_1 = e^{A(h-\tau)} \int_0^{\tau} e^{As} ds B \tag{31}$$
$$\Gamma_0 = \int_0^{h-\tau} e^{As} ds B \tag{32}$$
$$\tag{33}$$

The following section shows how a mapping between EFSM based functional description and the EQS based architectural description can be performed.

3.9 Mapping EFSM's on EQS'

The connection of a finite state machine with a generalised markov process can be done by a mapping. Doing this, we introduce the concept of time to a finite automaton. Every possible state input transition, i.e., every entry in the table of δ is assigned a value. The value depends on experience and measurements. It is, for example, the number of instructions a processor have to compute. Dividing the number of instructions by the instructions per time, we will get the processing time [2]. The number of instructions per time is normally measured in MIPS, (Million Instructions Per Second).

From the queuing theory point of view, we have a queue with n different types of customers. N is the number of the elements in the input set. Let M be the number of states. Then we can assign every element of the input up to M-different time values. The actually chosen depends on the current state. This is the service time μ for the i'th customer of type $e_a \in E_a = X_Z$ when the system is in state Z and signal X arrives.

$$X_Z = \delta(Z, X, P) \tag{34}$$
$$\mu(X_Z) = Proc_{time}(\delta(Z, X, P)) \tag{35}$$

The mapping of the eventset E of the Queuing System and the input alphabet of the EFSM is done by dividing the eventset in arrival E_a and departure E_d events. Every element of the cartesian product of the input and state alphabet, i.e., X_Z, of the EFSM is assigned a member of E_a. Additionally, there are state transitions of the EFSM which cause new events, (e.g. setting a timer causing an interrupt later in time). These additional events are also mapped to an element of E_a. To distinguish this event set from the input mapped event set E_a, we call this event set E_a. So we have

$$M : \delta(Z, X, P) \longrightarrow E_a \tag{36}$$

The set of departure events E_d are defined when a state-transition is served. For every service of an arrival event a departure event is generated after its associated transition. The time of the departure event relative to the system time is given by

$$Time_{e_d} = \mu(x_z) \tag{37}$$

When the departure event arrives the outputfunction of the EFSM fires:

$$P : \lambda(Z, X, P) \longrightarrow E_d \tag{38}$$

where $Proc_{time}(x_z)$ denotes the transition time for event X in state Z. Since for every event and state combination a different processing time is possible, we have for the ordinary FSM

$$|E_d| = |E_a| = |X \times Z| \tag{39}$$

whereas for the EFSM, Equation 39 is given by

$$|E_d| = |E_a| = \Sigma_1^{NOS} NOT \tag{40}$$

NOT denotes the number of transitions per state and NOS the number of states. Transitions can be fired by the arrival of an input event or an state event. Every arrival event has its own service time, so that for every specific arrival event e_a an explicit departure event e_d exits. This means, we have as many different customers as E_a events. Every event therefore has an assigned service time.

The state of the Queuing System is defined by the queued arrival event and its place in the queue.

$$H = [h_1, h_2, ...] = \{\delta(Z, X, P)\}^* \tag{41}$$

h_i defines the type of the event and i is the place in the queue. This situation is depicted in Fig. 6

Like the eventset, we can divide the statetransition of the Queuing System by arrival transitions and by departure transition. In the case of an arrival event we have

$$f_a(H, E_a) = \begin{cases} H \| \delta(Z, X, P) & \text{if} \mid H \mid > 0 \\ \\ h_1 & \text{if } \mid H \mid = 0 \end{cases} \tag{42}$$

The transition function for a departure event $e_d \in E_d$ is the same like Eq. 8 and Eq. 9 respectively. .

3.10 State Param Binding

In any system model, the state variables describe the dynamic state of the systems. However, when there are interacting systems, the state transition

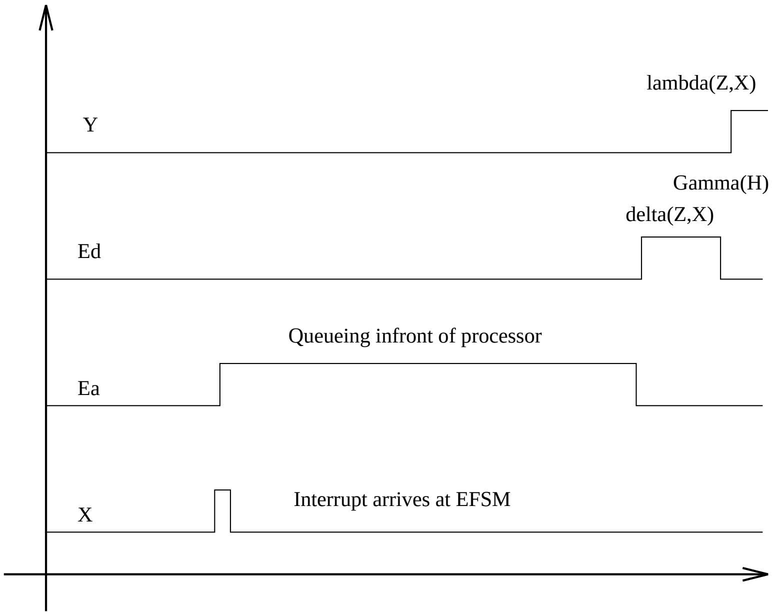

Figure 5: Timing scheme for coupling of an EFSM with an EQS

of system A depends on the state of a linked system B [13]. During A's transition, the state of B acts like a parameter:

$$Z^A_{new} = \delta^A(X, Z_{old}, Z^B) \tag{43}$$
$$= \delta^A(X, Z_{old}, P) withP = Z^B \tag{44}$$

This property can be used to exchange information between an EQS and an EFSM. For example, the state of a queue, i.e., its length, can be passed to an EFSM which decides whether to queue or to drop an arriving packet. In this example, it suffices to use the QS form of the EQS.

3.11 Coupled Systems

From the horizontal point of view, a system can be divided into two levels, i.e., the functional and the architectural level. The functional level is a network of EFSM's whereas the architectural level is a network of EQS. This is called horizontal grouping. To minimise the coupling links and to allow variable structure systems, we propose vertical grouping. We call this vertical grouping HiFAQ (Hierarchical Functional Augmented Queuing networks). The coupling between HiFAQ's carry information from the functional level as well as from the architectural level.

A HiFAQ is defined by:
Hif = $[I, O, SM, PSC, \Psi]$ where

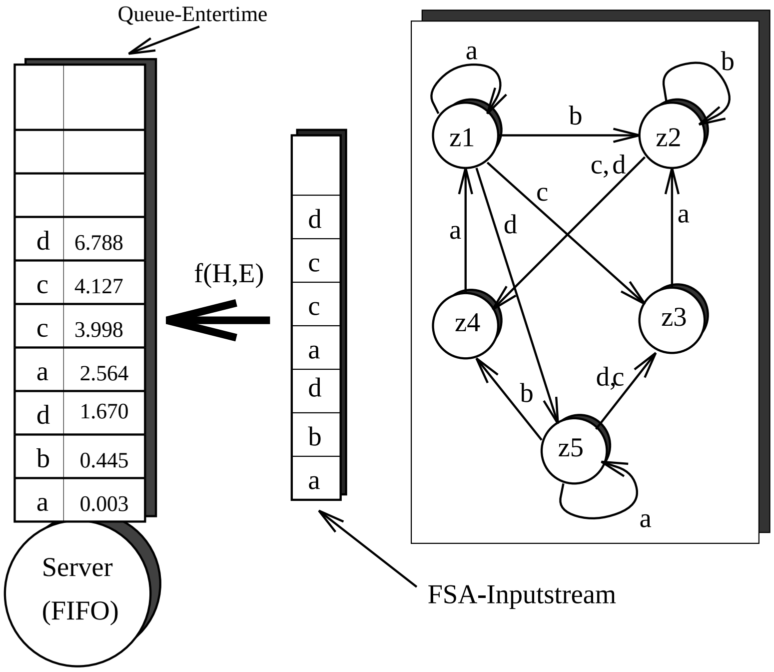

Figure 6: Mapping a functional description to an architecture

- I: Input ports
- O: Output ports
- SM: Submodels
- PSC: Parameter State Couplings
- Ψ: Coupling function between the SM's

A Submodel is defined by:
SM = $[I, O, M, \kappa]$ where

- I: Input ports
- O: Output ports
- M: Models $\{EFSM, EQN\}$
- κ: Mapping of the inputsignals to the input ports, the outputsignals to the output ports.

3.12 Inter HiFAQ communication

An overall system model can be divided into a network of HiFAQ's. A Model is defined as:

$$Model = [Hif, K_B]$$

K_B is the coupling between the HiFAQ's. K_B is a three dimensional Matrix. A two dimensional matrix describes which type of signals are sent between the HiFAQ's whereas the third dimension describes which HiFAQ's are coupled.

$$K_B[k,l,x] = \begin{cases} 1 & \text{if signal x of HiFAQ k} \\ & \text{ is sent to HiFAQ l} \\ 0 & \text{otherwise} \end{cases} \tag{45}$$

The functional level within a HiFAQ can consist of more than one EFSM. The same holds for the architectural level consisting of more than one EQS.

3.13 Intra HiFAQ communication

Almost the same scheme is used for the Intra HiFAQ communication. Each submodel, i.e, an EFSM or an EQS, has its own two dimensional matrix with the same format like the Ψ matrix of the HiFAQ's. However, to send a signal from one HiFAQ to another one, one has to couple this link from a submodel's output port to the HiFAQ's output port. The coupling has to be done on the input side of the receiving HiFAQ. Therefore, every HiFAQ has an additional two diminsional input mapping matrix. The output matrix of a HiFAQ represents a combination of the Ψ and κ matrix. The value of the κ matrix is the index to the Ψ matrix.

$$\Psi_j[o,x] = \Psi_k[i,\kappa[i,x]] \tag{46}$$

where $j,k \in Hif, i \in I, o \in O, x \in X$ in the case of an EFSM. The κ value hold the submodel SM of HiFAQ k.

4 The simulation class library

We developed a compiler for the HiFAQ language which translates HiFAQ models into C++ code. The C++ code uses intensively a class library capable of simulating continuous, time driven and event driven models simultaneously.

This is achieved by grouping the objects of the resulting simulation program into to categories. The first category consists of the active elements whereas the second one contains the passive elements:

- active objects: These objects are responsible for the time management. Simulation time is forwarded by events and integration algorithms. Hence, active objects are events and integration algorithms.

Active objects reacting to events are called discrete active objects whereas the objects doing integrations are named continuous objects The time between two consequetive events is submitted to the active continuous objects. While doing integration between the events, the time of the next event is monitored so that the system can react on state events.

- passive objects: Passive objects hold the functional description of the components as well as the inherent functions of the resources. These are finite state machines, differential equations, and queuing strategies. These functions are triggered by active objects, i.e., receiving a message from them.

Continuous active objects are maintained in a queue with its associated step time. The structure of the event/integration objects is depicted in in figure 7

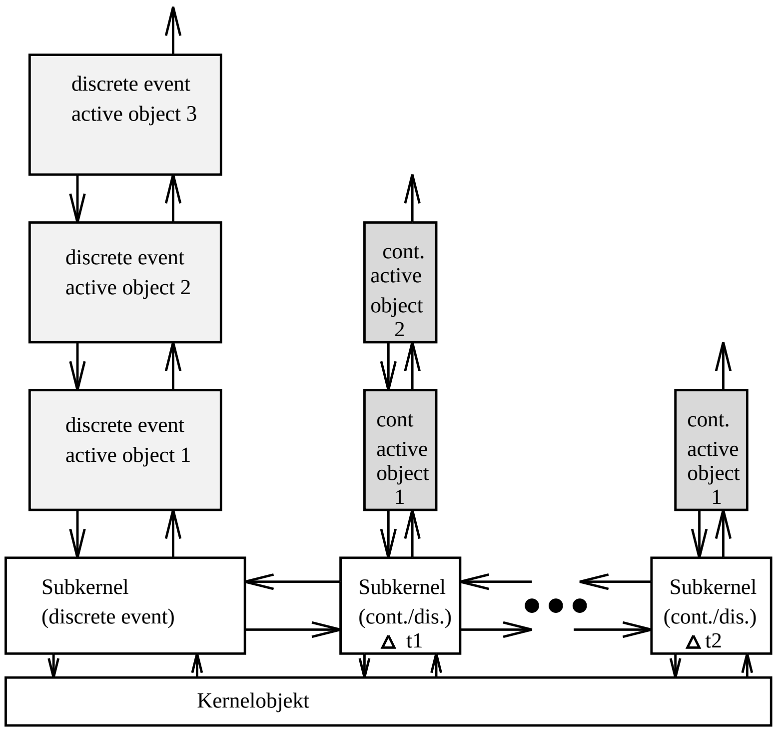

Figure 7: Kernel-Object structure

5 Example

The modelling and simulation of a simple 'Fly-by-Wire' control system by means of the HiFAQ description is shown. This system consists of a control loop implemented via a Token-Bus system. Such systems are common in automotive electronics [11]

A servo generates a signal R. Samples of this signal are transmitted to the regulator. The regulator also receives measurement values Y from the plant. Both values are compared and an error signal E is computed. The regulator calculates an actuator value U according to some control law. This value is transmitted to the plant. The plant sends a measurement value back to the regulator.

Assuming that the controller, the actuator, and the sensor are located at different locations connected via a bus system, and also assuming that the bus is also used by other stations, one comes up with the model of figure 8. From the control theory point of view it is a real-time system with random time delays [10].

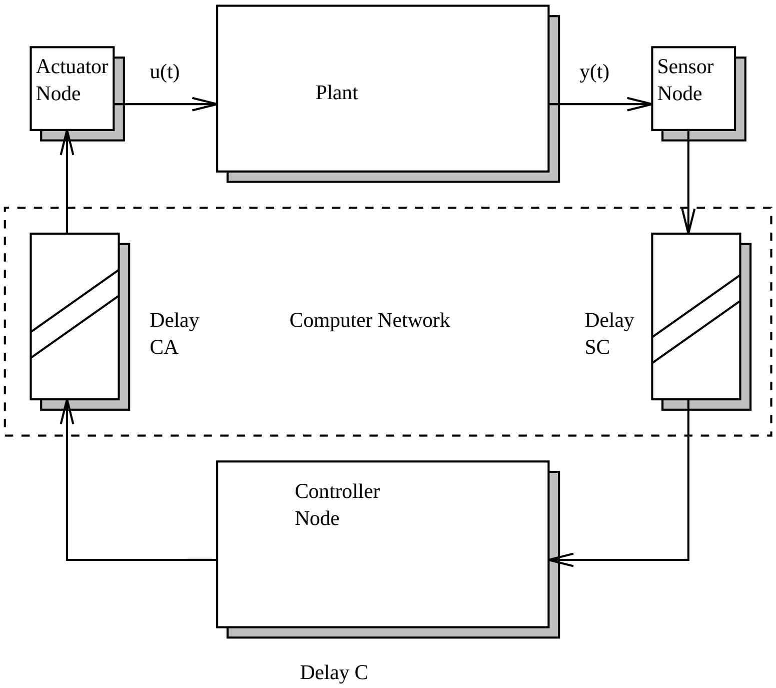

Figure 8: model of a control loop coupled by a communication network

The plant and the regulator are described by second order ordinary differential equations:

$$
\begin{aligned}
\dot{x_p} &= \begin{bmatrix} -3.3081 & 0 \\ 1 & 0 \end{bmatrix} x_p + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u \\
y_p &= \begin{bmatrix} 0 & 1 \end{bmatrix} x_p \\
e &= r - y \\
\dot{x_r} &= \begin{bmatrix} 0 & -4 \\ 1 & 0 \end{bmatrix} x_r + \begin{bmatrix} -9.88 \\ -0.7473 \end{bmatrix} e \\
y_r &= \begin{bmatrix} 0 & 1 \end{bmatrix} x_r + \begin{bmatrix} 4.9179 \end{bmatrix} e \\
u &= y_r
\end{aligned}
$$

where x_p is the plant state variable, x_r the regulator state variable, u is the actuator signal and y the measurement signal. The signal e denotes the error signal.

Fig. 9 shows how a bad implementation of the controller can influence the stability of the control loop.

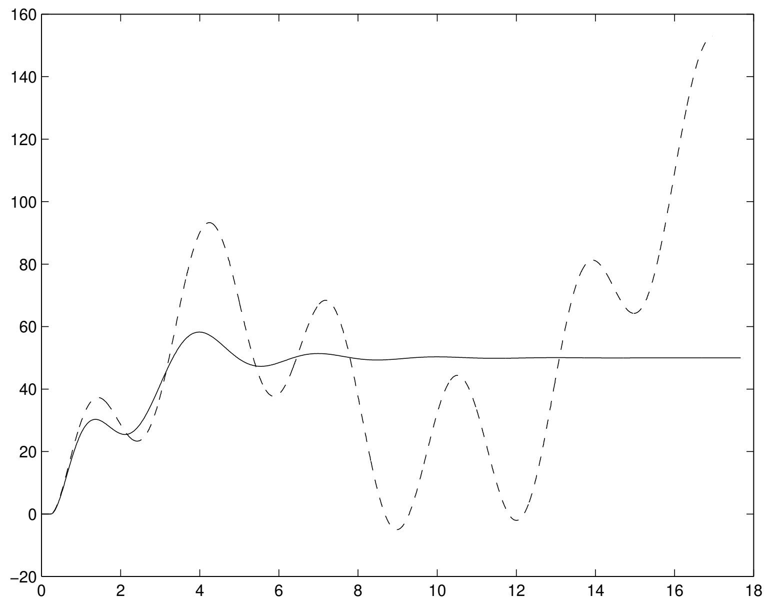

Figure 9: step response for the original (solid line) and a uncarefull event driven implementation (dashed line)

With a well fitted architecture one can achieve a better performance of the overall system. This case is shown in the next figure where the control and measurement values are used as early as possible for controlling the plant. Fig. 10 shows how a bad implementation of the controller can influence the stability of the control loop.

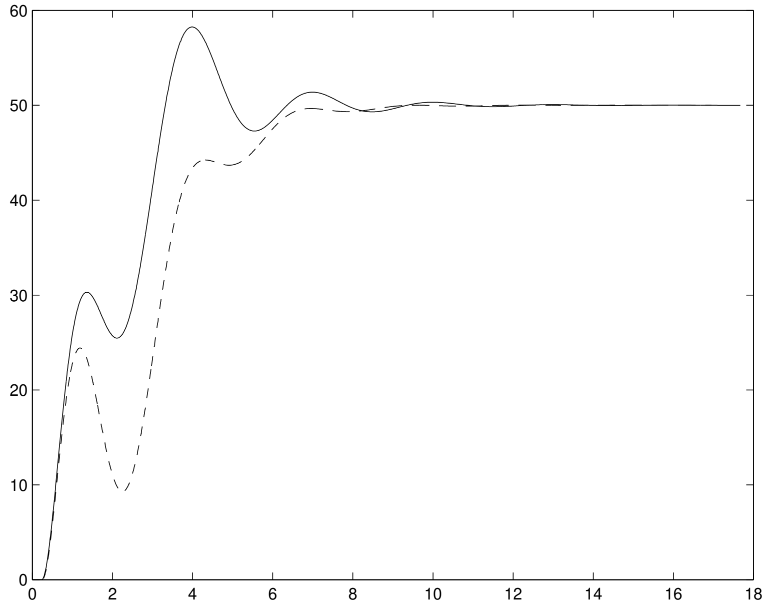

Figure 10: step response for the original (solid line) and a well fitted event driven implementation (dashed line)

6 Conclusions

In this paper we presented a modelling and simulation language for the integrated timed simulation of heterogeneous embedded systems. On a simple fly-by-wire control system the effects of architectural changes are unveiled.

Further investigations will be a database of computation time of several standard elements, the development of a framework HiFAQ's as well as the the development of control algorithms capable of dealing with variable time delay.

References

[1] R. Gerlich, *Early Validation of System Design by Behavioural Simulation*, ESTEC 3rd Workshop on 'Simulation for European Space Programs', Noordwijk, 1994

[2] C. Passerone et al. *Evaluation of trade-offs in the design of embedded systems via co-simulation* Technical Report, University of California at Berkeley, Berkeley 1996

[3] FZI,*Abschlussdokumentation des Projekts MSR* FZI, Karlsruhe 1996

[4] J. Ernst *Studie zu Spezifikation und Aufbau einer Cosimulationsumgebung unter der Ber"ucksichtigung des Simulator-Backplane Standards der CAD Framework Initiative*, FZI, Karlsruhe 1996

[5] J. Ernst, Y. Tanurhan, *Schnittstellen zwischen dem Entwurf und der Simulation in MESA und dem Prototyping in Merlan*, FZI, Karlsruhe 1996.

[6] W. Herden, H. Hillner, R. Mayer *Entwicklung, Modellbildung und Simulation in der MSR-Entwicklungsumgebung MESA*, Proceedings der ASIM 94, Stuttgart.

[7] U. Freund, *Moosim, a class library for mixed simulations*, Technical Report TU-Ilmenau, Ilmenau 1996

[8] U. Freund, *HiFAQ's, a description technique for the Virtual Design*, Technical Report TU-Ilmenau, Ilmenau 1996

[9] J. Bortolazzi, *Untersuchungen zur rechnergest"utzten Erfassung, Verwaltung und Pr"ufung von Anforderungsspezifikationen und Einsatzbedingungen elektronischer Steuerungs- und Regelungssysteme*, Dissertation, Universitaet Erlangen-Nuernberg, April 1994

[10] J. Nilsson, B. Bernhardsson, B. Wittenmark *Stochastic analysis of real-time systems with random time delays* Proc. 13.th IFAC World Congress, 1996

[11] U. Kiencke, T. Kytoelae, K.J. Neumann *Architectural Trends in Automotive Electronics* 1st IFAC-Workshop on Advances in Automotive Control, Ascona, 1995

[12] C. Cassandras *Discrete Event Systems*, Asken, Homewoold 1993

[13] M. Krapp *Digitale Automaten* Verlag Terchnik, Berlin 1988

A Example HiFAQ

The next lines of code describe a HiFAQ with a plant model, a network controller, a sampler, an actuator and its architecture. The EFSM are of Moore tyoe.

```
hifaq cplant  (destony:hifa,sampletime:real,meanlen:real)
input frombus;
state thestate;
output tobus;
parambind:
 plantstation.ql2->modactqueue.queuelen();
 plantstation.queuelen->pframequeue.queuelen();
 pgetframe.queuelen->pframequeue.queuelen();
 psample.yval->intplant.yhlp;
 intplant.u->plantstation.urec;
 plantstation.urec->intplant.u;
 intplant.bu->pactuator.btu;
 pactuator.ql2->modactqueue.queuelen();

models:

fsa plantstation (pprocessor)
input frombus: token,frame;
input fqueue    : frame;
input internalp: tokenend,frameend,tokenstart,framestart,framekill;
state thestate: waiting,sendframe,sendtoken,endtoken,endframe,purgeframe,
      repframe,fsr,tsr,rendframe;
output internalp : framestart,tokenstart,tokenend,frameend,framekill;
output tobus: token,frame;
initstate: waiting;
param: ql2,queuelen,urec;
hvars:
  destadr : int;
  aframe : frame;
delta:
   waiting -> sendframe : token & (queuelen > 0) TP 0.0000122;
   waiting -> sendtoken : token & (queuelen == 0) TP 0.0000177;
   sendframe->endframe : framestart TP respframe.length;
   sendframe->fsr : frame TP 0.00001;
   fsr->endframe : endframe TP 0.00001;
   sendtoken->tsr: frame TP 0.00001;
   tsr->endtoken : endtoken TP 0.00001;
   endframe -> sendtoken: frameend TP 0.00001;
   endtoken -> waiting: tokenend TP 0.00001;
   sendtoken -> endtoken: tokenstart TP 0.0000222;
   waiting -> repframe : frame &  (!myframe(frame.dest)) TP 0.0000122;
   repframe -> rendframe : framestart TP respframe.length;
   rendframe -> waiting : frameend TP 0.00001;
   waiting -> purgeframe: frame & (myframe(frame.dest)) TP 0.00001;
```

```
   purgeframe -> waiting : framekill TP 0.00001;

mue:
   sendtoken:  tokenstart set {thetoken = r_token;};
   sendframe:  {aframe = frame->fromqueue(pframequeue);} frame;
   sendframe: framestart set { respframe = aframe ;};
   repframe : {aframe = r_frame;} frame ;
   repframe: framestart set { respframe = aframe;};
   rendframe: frameend set {respframe = aframe;};
   endframe: frameend set {respframe = r_frame;};
   endtoken: tokenend set {thetoken = r_token ;};
   endtoken:  token;
   purgeframe: {
                urec = r_frame.value;
                frame->toqueue(modactqueue);
                probe uvalue urec;
               } framekill set {respframe = r_frame;};
ende fsa;

fsa pgetframe  (pproc2)
input frombus: frame;
input internal:queind;
input fromgen: frame;
state thestate: waiting,queueing,error;
output tqueue: frame;
output internal: queind;
initstate: waiting;
param : queuelen;
hvars:
actframe : frame;
delta:
   waiting -> queueing : frame TP 0.00003;
   queueing -> waiting: queind TP 0.00003;
   queueing -> error : frame TP 0.00001;
   error -> waiting : queind TP 0.00001;
mue:
   queueing: {frame->toqueue(pframequeue); } queind set {respframe = r_frame; };
   error: queind;
ende fsa;

fsa pactuator (pproc3)
input actint: next,frame;
state thestate:putvalue,idle;
output actint: next;
initstate: idle;
```

```
param: btu , ql2,queuelen;
hvars:
kaaf : frame;
at : real;
delta:
 putvalue -> putvalue: next & (ql2>0) TP 0.05;
 putvalue -> idle: next & (ql2 == 0) TP 0.05;
 idle -> putvalue:  next & (ql2>0) TP 0.05;
 idle -> idle :  next & (ql2 == 0) TP 0.05;
mue:
  putvalue: {
            kaaf = frame->fromqueue(modactqueue);
            btu = kaaf.value;
            probe queval btu;
          } next set  {tstation=parent; };
  idle: next set {tstation=parent;};
ende fsa;

qs pframequeue
input tqueue: frame;
state myqueuelen;
output fqueue: frame;
param: void;
delta:
   fifo;
mue:
  fqueue: frame;
ende qs;

fsa psample (pproc3) (destony:hifa,sampletime:real, meanlen:real)
input internals: next;
state thestate: nextsample;
output internals: next;
output fromgen:frame;
initstate: nextsample;
param: intar,queuelen,yval;
hvars:
  mlen :real;
delta:
  nextsample -> nextsample:    next TP sampletime;
mue:
nextsample:  {
                 source = parent;
                 length = meanlen;
                 dest = destony;
```

```
                 value = yval;
                 timestamp = tnow;
         } frame  ;
nextsample:      {tstation = parent;} next;
ende fsa;

qs modactqueue
input foproc: frame;
state myqueuelen;
output toproc: frame;
param: void;
delta:
   fifo;
mue:
  toproc: frame;
ende qs;

qs pprocessor
input internalp: framestart,tokenstart;
state queuelen;
output internalp: frameend,tokenend;
param: void;
delta:
  fifo;
mue:
  internalp: frameend;
  internalp: tokenend;
ende qs;

qs pproc2
input internal: frame,queind;
state queuelen;
output internal: queind;
param: void;
delta:
  fifo;
mue:
 internal: queind;
ende qs;

qs pproc3
input internals: next;
state queuelen;
output internals: next;
param: void;
```

```
delta:
  fifo;
mue:
 internals: next;
ende qs;

cont intplant
input fromsensor: u;
state  xa,xb;
output toactuator: y;
param: yhlp,y,bu;
dt : 0.05 ;
f:
xa = -3.3081 * xa + bu;
xb =  xa;
g:
y = xb;
yhlp =y;
ende cont;
ende hifaq;
```