A Model for Buffer Exploration in EDF Scheduled Embedded Systems

Frank Slomka
Department of Computer Science
University of Oldenburg
Ammerländer Heerstraße 114-118
26111 Oldenburg, Germany
slomka@informatik.uni-oldenburg.de

Jürgen Teich
Department of Computer Science 12
University of Erlangen-Nuremberg
Am Weichselgarten 3
91058 Erlangen, Germany
teich@informatik.uni-erlangen.de

Abstract

The present paper describes an extension to real-time analysis of earliest-deadline first (EDF) scheduled embedded systems to allow design space exploration with different intertask communication. The event stream approach for real-time analysis is generalized to unbuffered task communication. Adding this analysis technique to a well known cost model for system level synthesis, it becomes possible to explore different communication architectures of embedded systems by existing optimization algorithms.

1 Introduction

Real-time analysis is an important task during the design of an embedded system. Such a system must guarantee that the occurrences of computational results are in a timed relationship to the context in which the system is embedded. Examples are time slotted communication systems, automotive controllers and fly-by-wire systems.

An other important aspect is the analysis of different possible system implementations to find an minimal solution for the three main design parameters time, cost and power. Such an design space exploration can be supported by different optimization techniques to define a separate system level synthesis step during system design.

In the last years a lot of work considering system synthesis for real-time systems was published [1], [3], [4], [5], [9] and [13]. To find cost optimal solutions in embedded real-time systems, real-time analysis is combined with standard optimization techniques. The quality of the system synthesis approach depends on the used real-time analysis algorithm.

In embedded systems often a heterogeneous multiprocessor architecture is implemented. If data dependencies between tasks are needed to be considered, the original approach given by Liu and Layland [10] is not adequate anymore. Additionally, in multiprocessor systems some anomalies happen, e.g. a system with a feasible schedule is not correct anymore if the number of processors increase or the execution time of tasks would be decrease [8].

Because of this fact we have not only to deal with the worst case execution time (*WCET*) of tasks but also with their best case execution time (*BCET*) ([12] and [13]). In [13] an analysis method for this problem is described. To consider *BCET* and *WCET* leads to a state space explosion during the analysis of the system [12].

However, all previous discussed approaches are only dealing with periodic system stimuli, but real systems have to react on aperiodic stimuli, too. As it is shown in [8] aperiodic events can also be mapped on periodic server tasks. To describe all kinds of different system stimuli in a formal way, Gressers event stream model was developed [7] (see also [6]). Gresser defines an analysis technique for EDF scheduling. Because data dependencies easily can be resolved in EDF scheduled systems, the approach is a good candidate to analyze heterogeneous multiprocessor systems. Gressers model is limited to buffered task communication to make the system analyzable. The approach described in [11] also includes buffers to the systems implementation only to make real-time analysis possible. But such an approach leads to cost intensive implementations.

After giving a short introduction to the event stream model and Gressers analysis technique, an extension to analyze unbuffered task communication is presented. It is shown how this extension can be integrated into a general methodology for design space exploration of embedded systems. A short case study shows how the approach works.

2 Real Time Analysis

2.1 Event Streams

The event stream model describes the worst case timing relationships between events. The idea is to define a

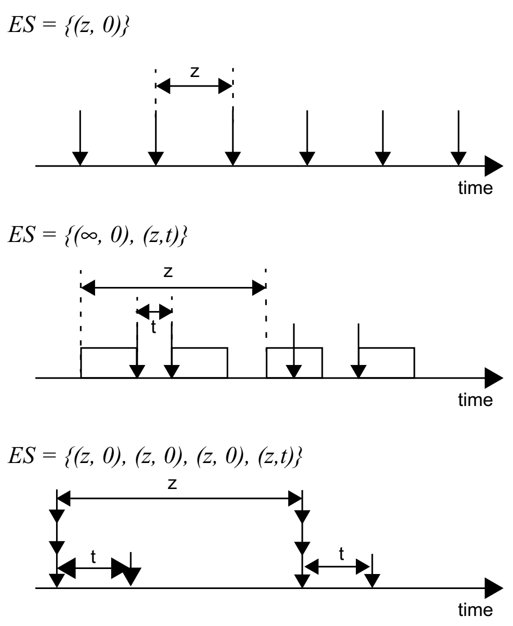

Figure 1: Event Streams [7]

minimal distance in time between one, two, three or more events in a formal specification of input stimuli. The model defines the maximum number of events in different given time intervals. Each event stream consists of a fixed number of event tuples:

$$ET = \begin{pmatrix} z \\ a \end{pmatrix} \qquad (1)$$

with a given cycle or period z and an interval a. An event stream is then defined as an ordered set of event tuples with a fixed meaning of each tuple:

$$ES = \{ET_i\} = \left\{\begin{pmatrix} z_i \\ a_i \end{pmatrix}\right\} \qquad (2)$$

The meaning of each tuple is defined by its position in the event stream. In this notation $i \in N$ represents the number of events which occur in the interval a_i. This means a_2 is the minimal distance between two events, a_3 the minimal distance between three and a_n is the minimal distance between n events. The reader may be confused, but in this notation the minimal distance between one and one event has to be defined, too.

Consider now the examples given in Fig. 1: The first example describes a periodic event as known from standard real-time analysis. The minimal distance between one and one event is *0* and the period of the stimuli is *z*. The second example shows a periodic event which jitters. The box in the figure gives the time interval in which the event jitters. This interval is not interesting for the analysis algorithm, because an event stream has to describe the worst case of timing relationships of events. However, the minimal distance between to events is *t*. The interval *t* occurs if an event occur at the end of the jitter interval and the next event is released at the beginning of the jitter interval. The replication period is also *z*. Note that $z = 2T$ if T is the period of the occurrence of the jitter boxes. The last example shows the release of more then one event at one point of time. The minimal distance between one, two and three events is 0, the minimal distance between four events is *t*. Note that these formalism only considers the worst case scenario. It is not necessary that all three events will occur at the same time, but it could be. Within this model it is also possible to describe aperiodic tasks. In such a case the period of the task is infinite as shown in example two by the first event tuple.

2.2 Real-Time Analysis

In this section the idea of real-time analysis with event streams is formulated. For a detailed formal discussion see [7].

The algorithm to analyze the real-time behavior of an embedded system depends on the scheduling strategy of the operating system. Rate Monotonic (*RMS*) and deadline monotonic (*DMS*) are well known strategies to schedule real-time tasks on one processor. However, the utilization of the processor is very bad if these algorithms are used. A strategy with high processor utilization is earliest deadline first (EDF). Using EDF scheduling the utilization of the processor is equal to *100%*. In terms of scheduling analysis this means, if the utilization $U \le 1$ the real-time system satisfies all its deadlines [10].

Gresser uses this equation as a starting point to define an analysis algorithm for event streams. The idea of the analysis algorithm is to construct an event function *E(I)* from the event stream and to use this function to build a utilization function of the system. The function *E(I)* describes the number of events for a given event stream in dependency of a given interval length *I*. If it is assumed that each task has a worst case execution time *c*, it is possible to derive the utilization function *C(I)* directly from the event function: Each event triggers a task with execution time *c*. This means that $C(I) = c$ in the interval *I*.

Consider the example in Fig. 2: The gantt chart shows three periodic tasks scheduled by *EDF*, each task with a worst case execution time *c = 2 time units* (*T.U.*). The utilization function is given by the dotted line on the right side of the figure. An interval of *0 T.U.* consists 1 event and a request of 2 *T.U.,* and an interval of *1 T.U.* a request *4 T.U.* to the processor of the system. Each task has a deadline of *3 T.U.* as shown by the boxes in the gantt diagram. The deadline of a task moves the utilization function of a task to the right. In the given example the utilization function moves 3 T.U. to the right.

For each task triggered by an event stream a separate utilization function has to be constructed, because different tasks may have different worst case execution times and deadlines. However, the sum of all utilization functions gives the utilization of the whole system.

In *EDF* scheduled systems the condition $U \le 1$ must hold. This means that the utilization function of the system shall be always under the bisecting line as shown in

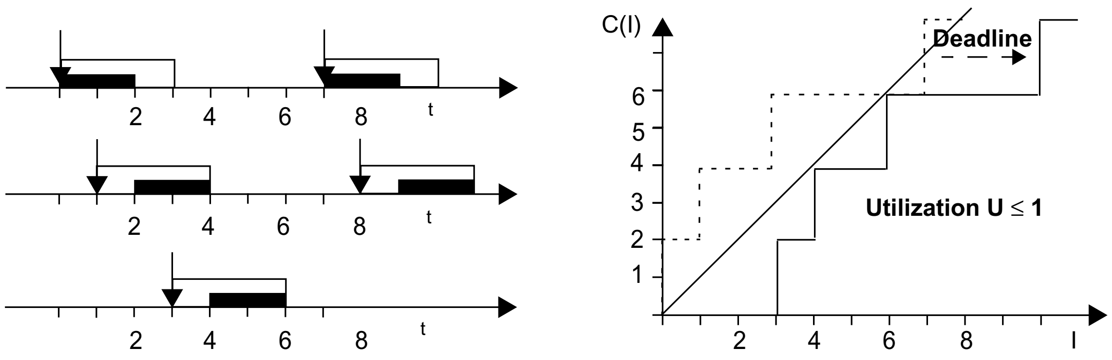

Figure 2: Real-Time Analysis [7]

Fig. 2. The analysis algorithm given in [7] has to check if *C(I)* is always smaller or equal to the bisecting line for all intervals I.

2.3 Data Dependencies

The definition of the analysis of data dependencies between tasks is easy in EDF scheduled applications. Data dependencies between tasks should not considered anymore during analysis, if the deadline d_2 of the successor tasks is modified [7][8]: The new deadline of the successor is the sum of the deadline of the predecessors deadline and the original deadline of the successor as shown in Fig. 3. It is now assumed that each of the booth tasks can be triggered by the same event stream. The modification of deadlines leads automatically to a correct schedule: The predecessor is scheduled by EDF before the successor, because of the earlier deadline.

The assumption that pre- and successor are triggered by the same event stream only holds if the communication between the two tasks is buffered. To proof this, just consider the case that the first instance of the predecessor task finishes after its worst case execution time and the second instance of the same task finishes after its best case execution time (*BCET*). Now, the resulting event stream used as input for the successor task has shorter worst case intervals than the originally event stream triggering the predecessor. All given minimal intervals shrink. In other words, a buffer between two communicating tasks restores the original minimal distance between different events of a event stream for the analysis.

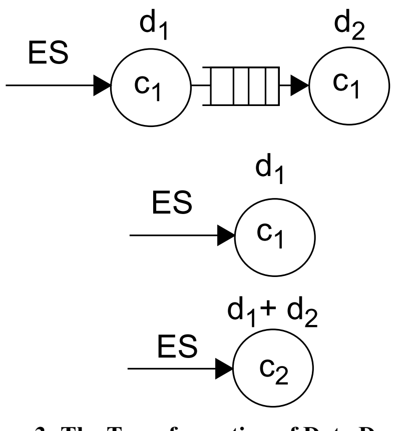

Figure 3: The Transformation of Data Dependencies [7]

3 Extensions for Design Space Exploration

3.1 Unbuffered communication

To consider buffer less communication between tasks, the event stream for successors has to be recalculated. For each event tuple a new minimal distance interval and period is needed.

Fig. 4 shows how a new interval a_i' and a new period z_i' are calculated. The minimal distance between two events is a_i. In a worst case situation the task instance triggered by the first event finishes after its worst case response time and the second task instance triggered by the next event of the event stream finishes after its *BCET* b_i. The response time of a task is the sum of the execution time of the task and the overall waiting time for tasks with higher priorities. However, the analysis algorithm does not calculate the response time of the tasks. On the other hand, a violated deadline results in an unfeasible schedule and it is possible to assume that the worst case response time of a task is equal to its deadline d_i.The deadline is exactly the worst case response time in a feasible schedule. The task instance triggered by the second event is not interrupted by any higher priority task and so it finishes after the *BCET* b_i.

The interval x_i in Fig. 4 illustrates the situation that the first instance finishes after d_i and the second instance finishes after b_i. The interval x_i is easily calculated by simple geometric considerations given by Fig. 4:

$$x_i = a_i + b_i - d_i \qquad (3)$$

It is not sufficient to consider only the interval x_i to calculate the new minimal distance between events.

An other relevant interval is y_i: The last instance of the first period finishes after its worst case response time d_i and the first instance of the second period finishes after

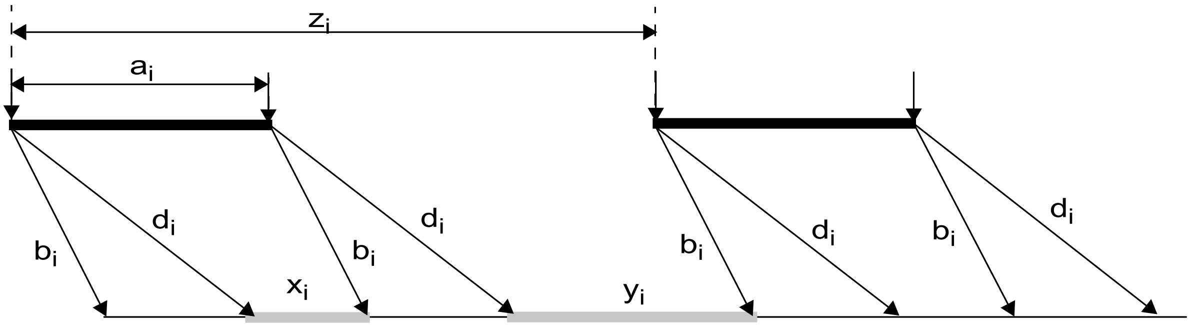

Figure 4: Calculation of the new Minimal Distance Intervals x_i and y_i

the BCET b_i. With the given period z_i the interval y_i is calculated by:

$$y_i = z_i + b_i - (a_i + d_i) \tag{4}$$

The new minimal distance of the modified event tuple a_i' is now the shortest distance of the two different cases:

$$a_i' = min(|x_i|, |y_i|) \tag{5}$$

It could be easily seen that the period z_i must not be recalculated in the most cases. To proof this, assume that $x_i < y_i$. Then, in the worst case, a new x_i occurs after $z_i + d_i$. The new period z_i' is then given by

$$z_i' = -d_i + z_i + d_i \tag{6}$$

In the other case, if $x_i > y_i$ then an new y_i occurs also after z_i:

$$z_i' = -d_i - a_i + z_i + a_i + d_i \tag{7}$$

Only if $x_i = y_i$ the period has to be recalculated. Then a new y_i occurs after

$$z_i' = x_i - b_i + d_i \tag{8}$$

and a new x_i after

$$z_i' = y_i - b_i + d_i \tag{9}$$

Eliminating x_i in equation (8) by using equation (3):

$$z_i' = (a_i + b_i - d_i) - b_i + d_i \tag{10}$$

or

$$z_i' = a_i \tag{11}$$

and substituting y_i in equation (9) with equation (4) leads to

$$z_i' = z_i + b_i - (a_i + d_i) - b_i + d_i \tag{12}$$

Using

$$z_i' = z_i - a_i \tag{13}$$

it could be shown that if $x_i = y_i$ then $z_i = 2a_i$ and therefore $z_i' = a_i$.

To support a full design space exploration an additional cost model is needed. The graph model introduced in [2] is used as a general approach for system synthesis. In the following section this model is described and then an extension for buffer analysis is introduced.

3.2 Model for Design Space Exploration

The model discussed in [2] consists of a specification graph, which is split in two main parts, a problem graph to model the behavior of the systems algorithms and an architecture graph to describe the hardware architecture of the system.

The problem graph is composed of two types of nodes, nodes to model tasks and nodes to model communication. Edges in this graph are describing data dependencies between the different tasks of the application.

The architecture graph also consists two different types of nodes, processing elements as resources for computation, and buses to perform communication between the computational elements. Each node of the architecture graph is annotated by a number. This number describes the cost of the node, or, the cost of the given hardware element. This fixed cost may be interpreted as

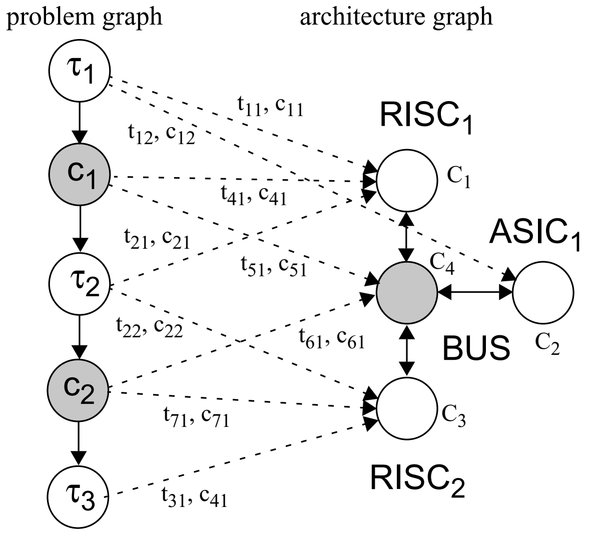

Figure 5: System Specification

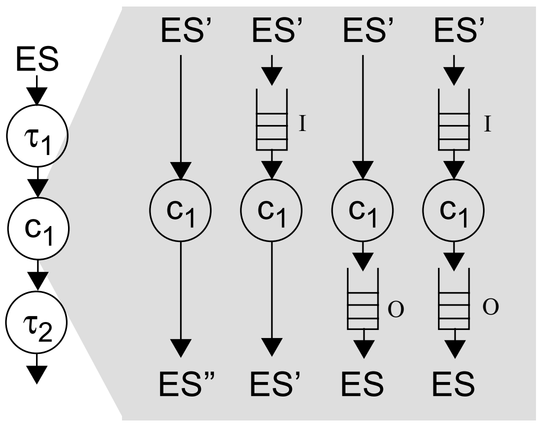

Figure 6: Possible Implementations of Communication

the usage of silicon area of a processing or communicating element.

The problem graph and the architecture graph are connected by binding edges. Each edge describes a possible binding of a function of the problem graph to a hardware resource. The two numbers annotated to a binding edge are describing the time consumption of computation or communication on the related hardware resource and an additional cost. This additional cost models the consumption of memory to perform the function of the related software task on the hardware.

A genetic optimization technique is used to explore different allocations of hardware resources and different bindings of problem graph nodes to architecture graph nodes. After optimization each problem graph node is related to exact one architecture node and the overall system costs are minimized [2].

3.3 Buffer Analysis

To explore the design space of buffered and unbuffered communication a few modifications of the discussed model are necessary.

To each binding edge of the specification graph a *WCET* and a *BCET* has to be assigned. Each computation node of the problem graph, or each task, additionally has a deadline annotated when performing *EDF* scheduling. To each edge of the problem graph an event stream is added. The event stream at the beginning nodes of a path of the problem graph are needed to be specified by the system designer and all event streams of successor nodes will be calculated by the analysis algorithm as described in Section 3.1.

The most important modification of the model is the extension of the cost model of communication nodes. As seen in Fig. 6, four different implementation strategies of one communication node are possible. It is possible to implement a buffer before and/or after a communication node. No modifications of the event stream are necessary if both buffers are implemented in the final system. In all other cases, one or two modifications of the successor event streams must be calculated.

The buffer in front of the communication node is called the input (I) buffer while the buffer after the communication node is called the output buffer (O). Each buffer implementation needs chip area. This is modelled by defining a modification of the binding costs of the edges of the specification graph.

The costs of a buffer are depending on the number of messages maximal stored in the buffer. To compute the number of maximal events stored in a buffer, the number of events which occur during the maximum storage time - the interval δ - of a event stream is calculated [7]:

$$n = ES(\delta) \tag{14}$$

For input buffers the maximum length of this interval is given by the difference between the deadline of the communication task and the *BCET* of the sender task

$$n_I = ES(\delta_I) = E(d_c - t_{s,b}) \tag{15}$$

The number of events stored in an output buffer is calculated as follows:

$$n_O = ES(\delta_O) = E(d_e - t_{r,w} - t_{s,b}) \tag{16}$$

with d_r as the deadline and t_{rw} as the *WCET* of the receiver task. The modified binding costs of a communication node are modeled by using the constant c_{buf} which describes the costs to store exact one event message in the buffer:

$$c_b' = c_b + (n_I + n_O)c_{buf} \tag{17}$$

This equation substitutes the original binding cost c_b annotated to the binding edge of the communication node. In other words, the cost of a buffer is the number of the maximal stored messages multiplied with the cost to store exactly one message. Such a calculation of buffer costs is performed by the real-time analysis algorithm, while the allocation of buffers is done by the optimization technique itself. To allow the optimizer the access to the buffer model, two binary variables are added to equation (17), each as a multiplier for n_I and n_O. An other way is to set n_I and n_O directly by the optimizer and to recalculate all terms n_I and n_O unequal zero by the analysis algorithm.

4 Case Study

To illustrate the results of the previous sections a simple example is discussed in the following: Consider a network router for digital telephone systems, a switch for ATM or a router for quality of service connections in the internet (QoS). In Fig. 7 a possible problem graph for such a router is shown. The processes P_i are modelling the input protocol functionality of the incoming lines. The process C_1 is modelling the overall control function, while the processes S_i are modelling the scheduler for the QoS output lines. Communication is performed by the tasks K_i. The event streams of Table 1 are given, if it is assumed that P_1 is connected to a SONET/SDH line

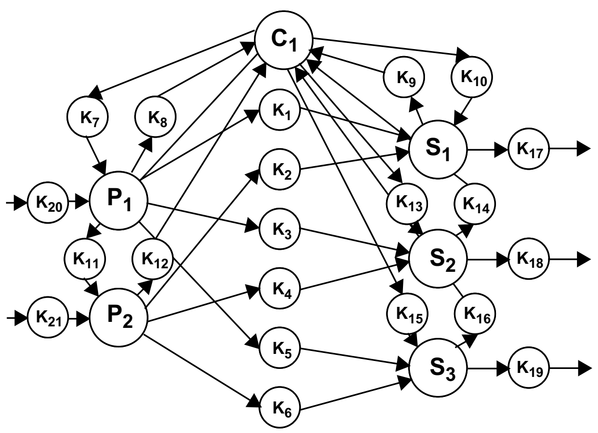

Figure 7: Problem Graph ATM-Router

and P_2 is connected to an ATM line. Furthermore it is assumed that only one processor type and one bus resource is available to implement the system. The processes execution times and deadlines are given in Table 2.With this informations it is possible to calculate ES' and ES'' as shown in Table 1.

ES	ES'	ES"
{(0.125 µs, 0)}	{(0.125 µs, 0.25 µs)}	{(0.125 µs, 0.25 µs)}
{(2.78 µs, 0)}	{(2.78 ms, 0.78 ms)}	{(2.78 ms, 0.22 ms)}

Table 1: Event Streams

Task	BCET	WCET	deadline
P_1	0.5 µs	1 µs	2 µs
P_2	2 ms	2 ms	4 ms
S_i	1 ms	2 ms	4 ms
K_i (P_1)	0.5 µs	0.5 µs	1 µs
K_i (P_2)	1 ms	1ms	2 ms

Table 2: Execution Times and Deadlines

The system could be implemented by different hardware architectures: 1) All processes P_i are implemented on one processor or 2) each P_i is implemented on an exclusive processor. It is also possible to implement the processes S_i on 3) one processor or 4) on three different processors. The design space for system level synthesis contains all combinations of 1) to 4).

However, if it is assumed that each communication took 1 ms the shortest interval length without implementing buffers becomes 0.78 ms. A real-time analysis using this value shows that all processes can be scheduled without implementing any buffer on one output processor.

This result holds only for three scheduler tasks. If just one output line is added to the specification, it is not possible to schedule all scheduler tasks on one processor anymore. Including buffers to the implementation solves this problem without changing the processor allocation.

The example shows that it is possible to explore the design space and to find out if it is better to implement faster and/or more processors or to include buffers by using the same number of resources.

5 Conclusion

The paper shows how a design space exploration of buffered and unbuffered communication is added to system level synthesis of real-time systems. The new method is based on a few extensions to an existing real-time analysis model. This model is combined with an existing synthesis approach for embedded systems.

The new concept of the paper is demonstrated by a short real live example. However, the next step is to implement the approach in an existing system synthesis framework to validate the results by different complex examples.

6 References

[1] J. Axelsson. *Analysis and Synthesis of Heterogeneous Real-Time Systems*. Dissertation, Linköping Studies in Science and Technology, 502. 1997.

[2] T. Blickle, J. Teich, L. Thiele. *System-Level Synthesis Using Evolutionary Algorithms.* Design Automation For Embedded Systems, Kluwer Academic Publisher, Boston, 3(1), 1998.

[3] Dave, B.P. und Jha, N.K. COHRA. *Hardware-Software Cosynthesis on Hierarchical Heterogeneous Distributed Embedded Systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 17(10). 1998.

[4] B.P. Dave, G. Lakshminarayana, N.K. Jha. *COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems.* IEEE Transactions on Very Large Scale Integration (VLSI). 7(1), 1999.

[5] R.P. Dick, N.K. Jha. *MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 17(10). 1998.

[6] F. Fischer, A. Muth, G. Färber: *Towards interprocess communication and interface synthesis for a heterogeneous real-time rapid prototyping environment.* 6th International Workshop on Hardware/Software Codesign, IEEE Computer Society Press. Seattle, USA, 1998.

[7] Gresser, K. 1993. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme.* Dissertation, VDI Verlag, Düsseldorf, 10(268).

[8] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publisher. 1997.

[9] C. Lee, ,M. Potkonjak, W. Wolf. *Synthesis of Hard Real-Time Application Specific Systems.* Design Automation for Embedded Systems. Kluwer Academic Publisher, Boston, 4(4), 1999.

[10] C. Liu, J. Layland. *Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments.* Journal of the ACM 20(1), 1973.

[11] K. Richter, R. Ernst. *Event Model Interfaces for Heterogeneous System Analysis.* DATE 2002, Design, Automation and Test in Europe, IEEE Computer Society Press, Paris, France, 2002.

[12] F. Slomka, J. Zant, L. Lambert. *Schedulability Analysis of Heterogeneous Systems for Performance Message Sequence Chart*. 6th International Workshop on Hardware/Software Codesign. IEEE Computer Society Press. Seattle, 1998.

[13] T. Yen, W. Wolf. *Hardware-Software Co-Synthesis of Distributed Embedded Systems*. Kluwer Academic Publisher, Boston. 1996.