Hierarchical Event Streams and Event Dependency Graphs: A New Computational Model for Embedded Real-Time Systems

Karsten Albers, Frank Bodmann, Frank Slomka
Department of Computer Science, University of Oldenburg
{first name, second name}@informatik.uni-oldenburg.de

Abstract

One of the important aspects of schedulability analysis is the model used to describe the system and its timing behavior. On one side, the accuracy of the test strongly depends on the accuracy of the model. On the other side, a detailed model could lead to an inaccaptable evaluation time. In this paper we propose a new model, the hierarchical event streams, which allows a high accuracy. We provide an efficient feasibility test for the model based on the context of demand and request bound function. Additionally we will provide a methodology to extract this model out of a control-flow graph efficiently. Together this allows a more accurate and efficient schedulability analysis of event driven real- time systems.

1. Introduction

Finding cost optimal hardware solutions and verifying existing hardware for hard real time systems requires an expressive real time model. Appropriate models are characterized by removing unnecessary information while maintaining the important essence. The model for real time analysis introduced in this paper takes the inner state of tasks into account, but abstracts from it by transforming the tasks control flow into its impact on the overall system within arbitrary time intervals.

An accurate description of the timing of events is key to an accurate determination of the load of the system and therefore to an accurate real time analysis. This in turn allows optimal hardware components to be chosen that still satisfy the desired timings. Events can be generated not only by the environment, for example by sensors, but also by tasks within the system. A single execution of a task can produce several events. For an accurate real time analysis it is essential to determine the exact timing behavior of these internal events.

In recent years a lot of work was published about real-time analysis. However, many of these techniques have an exponential run-time complexity. To avoid this approximative algorithms with linear complexity were developed in real-time schedulability theory. But most of the models of computations used for embedded system design are not suitable to be combined with real-time schedulability theory. To bridge this gap we suggest a new model for the schedulability analysis of distributed embedded real-time systems. This new model can automatically be derived out of the control flow graph, a data structure which is readily available during the design flow. The model regards the stimuli of the system as well as a description of the interprocess communication. The idea is to describe the communication behavior of the tasks by the same model as the systems stimuli. This internal communication causes dynamic task activations. See for example Figure 1 where the task τ_1 activates the tasks τ_2 and τ_3. These activations are caused by events that are generated during the execution of the task. Conventional models assume the generation of events to occur only at the end of a tasks execution. To capture this dynamic behavior a new model is required.

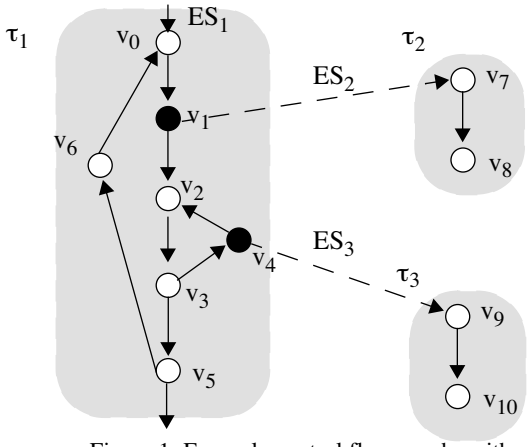


Figure 1: Example control flow graphs with dynamic task activation

First the existing event stream model is extended to the more expressive hierarchical event stream model. This mathematical description allows to elegantly characterize system stimuli which are containing both bursts and complex periodic / aperiodic behavior. In this context an efficient feasibility analysis for the hierarchical event streams is presented which is based on the concept of demand and request bound functions.

In the second part of the paper we provide the methodology to derive the new model out of the

** The research described is partly supported by the Deutsche Forschungsgemeinschaft under grant SL 47/1-1, SL 47/2-1*

system´s description given by control flow graphs. The new model can be acquired without an expensive state-based analysis. Our model is stateless and only based on a quantitative description of the temporal density of events.

2. Related work

A well known model of computation for data dominated systems are the synchronous data flow graphs (*SDF*). Synchronous data flow graphs are a rate-based execution model. Each node of the graph is annotated by the number of fired and the number of consumed tokens. The model has a close relationship to petri nets.

In the System-Property-Interval model (*SPI*) [10] embedded systems are described as coupled processes; input and output rates of the processes can be specified by intervals. It is not possible to use efficient feasibility tests on this model. Additionally it is difficult to use powerful event models like event streams. This would require complex mathematical transformations. However, the goal for system models should be an acceptable run-time of the applicable analysis algorithms.

An alternative attempt to connect the methods of the real-time analysis with token-based procedures is described in [14] and [16] by Goddard and Jeffay. In comparison to the event stream model [15] this description is however limited, because it is not possible to describe processes with multiple rates and jitter. The model also does not allow the description of overlapping executions of the same task. The work of Goddard and Jeffay shows how *SDF*s can be connected to scheduling theory. However, the work is limited to the periodic task model and not suitable for distributed systems with different processing elements connected by communication structures without buffers.

A very powerful model to describe distributed real-time systems was introduced by Thiele et al. [8], [9] and is called real-time calculus. The model works with approximated arrival curves. So the model is limited to an approximated analysis and can not be used for an exact one. In [7] Chakraborty and Thiele propose a model specifically designated for bursts. We will discuss this model in the evaluation.

To the best of our knowledge there are no integrated methodologies available so far which allow to extract the exact event stimuli of a task out of the control flow graph of the task activating that task if there is no restriction which nodes of the control flow graph may cause task activations.

In this work we will give such a methodology and also the schedulability analysis for systems with static or dynamic priorities.

3. Contributions

This paper provides several contributions. First, we propose the hierarchical event stream model as a new accurate and efficient model for the stimuli of real-time systems. It allows for the first time to model efficiently complicated stimuli including both bursts and aperiodic behavior. We give an efficient feasibility test for this model based on the concepts of demand and request bound functions.

In the second part we introduce for the first time the event dependency analysis. It is a methodology to calculate the timing relationships of events generated by a task. This approach is not limited to tasks that generate events at the end of an instance. Instead with the new approach tasks can be analyzed that generate events anytime during their execution. We propose a methodology to extract the timing relationship between these events out of the control flow graph of the tasks. This allows a precise description of the stimuli.

4. Model

The goal of the model used in this paper is to analyze the real-time behavior of an embedded real-time system. The question is, if it is possible to schedule all the needed jobs of the system in such a way that all deadlines of the jobs are met.

To achieve this goal the model needs a representation to describe the stimuli and the required computational time of the applications jobs. In the following we introduce the model to describe the structure of the application followed by the systems stimuli model. To model the stimuli of a task we introduce for the first time hierarchical event streams.

4.1. Task graph model

An embedded system consists of several different building blocks. On the hardware side there are processing elements, memories and communication components. On the software side we have program code and data structures. The application can be split up into parallel running processes. In this model we assume that each process does not itself contain further parallelism and can therefore easily be transferred into a control flow graph.

Def. 1: Control Flow Graph: *A control flow graph consists of a set of nodes representing the basic blocks of a program and a set of edges describing the control flow between the basic blocks.*

Using the control flow graph the static behavior of the program such as the latency under worst-, best- and average case conditions can be analyzed. Basic blocks that create events for other tasks are represented by marked nodes. These basic blocks create events at the end of each of their executions. A task can contain several of these marked nodes and they can trigger different tasks or all the same task. We allow loops and branches within the control flow. The maximum number of iterations of loops are bounded. Parallel or pseudo parallel running processes of the application can be modeled using task graphs:

Def. 2: Task: *A task τ is an execution path through a control flow graph. It is characterized by a 4-tuple $\tau = (c, b, d, \Theta)$ where c is the worst case execution time, b the best case execution time, d the relative deadline and Θ the event stream triggering the task.*

The definition for event streams follows in Section 4.2. For the purpose of this paper we keep the definition of tasks simple. Extensions for variable execution times can be easily included following the approaches given in [12],[13].

Def. 3: Task Graph: *A task graph is a directed graph. Nodes in a task graph represent the tasks of an application while the edges (τ_1,τ_2) describe that the task τ_1 can activate task τ_2. For activation from the environment the task graph may include additional source nodes.*

Activations of a task can result out of interrupts, data dependencies, signals etc. An event is a single activation. A task graph only describes the possibility of an activation.

For the analysis of real-time systems the timing relationship between the events is essential. A well known accurate model are synchronous data flow graphs (SDF). However, in SDFs no information is provided about the timing relationships of the events. This makes it difficult to use SDFs as a data model for the well known real-time feasibility analysis. To bridge the gap between the SDF model and the scheduling theory the event dependency graph is introduced:

Def. 4: Event Dependency Graph: *An event dependency graph is a task graph where the edges are additionally weighted by the temporal density of activations.*

We will propose a methodology to extract the incoming event streams for each task out of the control flow of the triggering task graph.

4.2. Stimuli and Intertask Communication Model

The triggering stimuli for the tasks are modeled using event streams. In contrast to the related work, we do not only allow events to occur at the end of tasks, but also within their executions. Each node of the control flow graph can generate events triggering other tasks. Considering this more accurate behavior is especially interesting for tasks generating more than one event. The model allows to represent the events in a more precise way. In this work we will provide a methodology to extract the exact outgoing event streams out of the control-flow graphs of the tasks. We will also provide a new model to describe event streams in a more precise and efficient way than with the existing models.

4.2.1 Event Streams

Event streams where first defined in [15]. The purpose was to give a generalized description for stimuli. The basic idea is to provide an efficient general notation for the event bound function. For every interval *I* the event bound function *EBF(I)* can calculate the maximum number of events which can occur within *I*. For this only the length of *I* is relevant, not a specific start and end point. In the following, when speaking of intervals we always refer to its length only.

The goal of the event stream model is to provide an efficient general notation for the event bound function. A general way to represent events is by the set of the distance of each event to a common start time. The distance can also be regarded as a time interval between the start time and the event. We will call this set an event sequence.

Note that it is not possible to represent an infinite number of events in this way. Therefore events are grouped into periodic sequences. Such a periodic sequence can be modeled by a single tupel consisting of a period and an interval which describes the distance for the first event of the sequence. An event sequence is a set of such tuples. Events which do not fit in a periodic behavior are modeled with an infinite period. Formally an event sequence Θ is characterized by a set of event elements ω where ω is a tuple *(p,a)* with *p* being the period and *a* the interval describing the initial distance. The set can include equal elements several times.

This formal description allows an efficient realization for the event sequence function.

Def. 5: Event Sequence Function: *An event sequence function* $ESF(I, \Theta)$ *provides the number of events occurring within the interval I located at the start of the sequence* Θ.

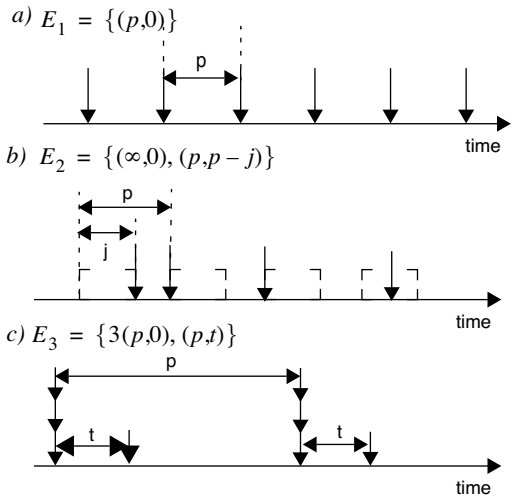


Figure 2: Event Streams [15]

Lemma 1: *The event sequence function for an event sequence* Θ *and an interval I can be determined by the following formula:*

$$ESF(I, \Theta) = \sum_{\omega \in \Theta} \left\lfloor \frac{I - a_\omega}{p_\omega} + 1 \right\rfloor$$

Proof: See [15]. □

The event sequence function always shows a monotonic non-decreasing behavior. As a shortcut for notation equal elements are represented only one time together with their quantity. An event sequence is called homogeneous if all event elements share the same period or have an infinite period. It is possible to transfer each event sequence into a homogeneous one by exchanging each period within the sequence by the least common multiple of the periods. To compensate this step it is necessary to insert additional event sequence elements.

For real time analysis it is necessary that the event bound function is a sub-additive function.

Def. 6: Event Stream: *An event sequence* Θ *is called an event stream if for all intervals I, J:*

$$ESF(I + J, \Theta) \leq ESF(I, \Theta) + ESF(J, \Theta)$$

That means the highest density of events occurs always at the start of the sequence. An event sequence for which the corresponding event seqeunce function fulfills this condition is called event stream. Each homogeneous event sequence can be transferred to an event stream by reordering the events (and recalculating the intervals).

Def. 7: Event Bound Function: *The event bound function* $EBF(I,\Theta)$ *calculates the maximum number of events which can occur within any interval of length I within the sequence* Θ.

Lemma 2: *If a sequence* Θ *is an event stream, the event bound function* $EBF(I,\Theta)$ *is equivalent to the event sequence function* $ESF(I,\Theta)$.

Note, that for the purpose of real-time analysis it is not necessary to model the event sequence first, because the event stream can be directly extracted out of the system description. In Figure 2 some examples for event streams

can be found. The first shows the event stream for a strict periodic stimuli. The second shows periodic stimuli in which the single events can jitter within a jitter interval of size *j*. In the worst case situation one event occurs at the end of a jitter interval and one event occures and the start of the following jitter interval. This leads to *p-j* as the length of the minimal interval containing two events. For three event the length of the interval is given if the first events occurs at the end of a jitter interval and the third event at the beginning of its jitter interval. The length is given by *p+p-j*. For four event the length is *2p+p-j* and so on. This behavior can be exactly modeled with E_2.The third example shows an irregular behavior, with the possibility for three events to occur at the same time and the fourth to occur after a time *t*. This structure can be repeated using a period of *p*. Obviously all these examples can be described by event streams in an easy and intuitive way.

In theory it is possible to describe each set of events by an event stream. For some sets of events the number of tuples needed for description can become quite large. Especially the description of bursts is inefficient because it is necessary to model each element of the burst with a separate tuple. For the purpose of evaluation it is not necessary to find the exact minimum intervals. It is sufficient to find for all intervals a lower bound. This can allow to simplify the event stream (also might mean to accept an overly pessimistic description).

A detailed definition of the concept and the mathematical foundation can be found in [2].

The inverse function for the event bound function is the interval bound function.

Lemma 3: *The interval bound function returns for a given number of events the minimum interval in which this number can occur:*

$$IF(n, \Theta) = min(I | (EBF(I, \Theta) = n))$$

For a homogeneous sequence the implementation of this function is easy. It is only necessary to calculate the last event sequence element and the number of completed periods.

4.2.2 Hierarchical Event Streams

The event stream model is a very general model. The problem is that the description for burst can become quite large. A burst consists of a number of events which occur within a short amount of time followed by a waiting period. Bursts can be the result of loops in the control flow graph of previous tasks. Each iteration of the loop produces one event. The number of iterations are bounded, therefore the number of events occurring as the result of one activation of the loop are bounded too. Together they form the burst.

In Figure 1 we present an example task graph to illustrate these bursts. τ_1 is triggered by a period event stream. It triggers two other tasks τ_2 and τ_3. The event stream for both tasks shows a bursty behavior. The control flow graph of τ_1 consist of two nested loops. In the outer loop v_1 generates the events triggering τ_2. v_4 is the node belonging to the inner loop which generates the events triggering τ_3. One event is generated in every iteration of the loop. Therefore the event streams ES_2 and ES_3 triggering τ_2 and τ_3 shows a bursty behavior, whereby the behavior of τ_3 is complicated due to the nested loops. The purpose of this paper is to present a model which is able to describe these event streams in an easy and efficient way. We will also present an efficient real-time analysis for this model.

The problem of the event stream model is that for each event of a burst an additional event sequence element is needed. For an efficient formal description of bursts we propose an extension of this model. We allow an event element to generate a set of events instead of just a single event. The new hierarchical event sequence elements are a 4-tuple:

$$\omega' = \begin{pmatrix} p & n \\ a & \Theta' \end{pmatrix}$$

where *p* and *a* are the period and the initial interval, *n* is the limitation for the number of events generated by this element during one period. Θ' is an embedded (hierarchical) event sequence providing the pattern for the generation of the events. If this pattern would generate more than *n* events only the first *n* events are considered. A hierarchical event sequence is either a set of event sequence elements or a single event *e*.

Def. 8: Separation Condition: *A hierarchical event sequence element* ω *fulfills the separation condition if*

$$IF(n_\omega, \Theta_\omega) \leq p_\omega$$

It should not be possible that the events which are generated within different periods can overlap. With this condition it is easy to extract the event bound function.

Lemma 4: Hierarchical Event Sequence Function: *For a hierarchical event sequence* Θ' *fulfilling the separation condition the event sequence function can be determined as follows:*

$$ESF(I, \Theta') = \begin{cases} 1 & \Theta' = e \\ \sum_{\varpi \in \Theta'} \left(\left\lfloor \frac{I - a_\varpi}{p_\varpi} \right\rfloor \cdot n_\omega + R(I, \varpi) \right) & else \end{cases}$$

$$R(I, \varpi) = min(n, ESF(((I - a_\varpi) mod\ p_\varpi), \Theta_\varpi'))$$

Proof: Due to the separation condition it is always possible to include the maximum allowed number of events for the completed periods. The calculation is corresponding to the event sequence function. Only the last possible incomplete period has to be considered separately. This incomplete period is calculated by the modulo operation between $I\text{-}a_i$ and p_i. The number of events which can be generated in this interval at maximum, can be evaluated by the event seqeunce function of the sub event sequence limited by the maximum allowed number of events for this sequence. If the sub event sequence consist of a single event only, the maximum number of generated events is one.□

As a short notation for a hierarchical event tuple with a single event we can use a normal event sequence element:

$$\begin{pmatrix} p & 1 \\ a & e \end{pmatrix} = \begin{pmatrix} p \\ a \end{pmatrix}$$

It is not necessary for the sequences to be homogenous.

The defintions of hierachical event streams and the hierachical event bound function is corresponding to the defintions in Section 4.2.1.

5. Feasibility Analysis

The feasibility analysis closely follows the analysis proposed by Baruah in [3] and therefore we keep its explanation short.

For our feasibility analysis we use the concepts of the demand and request bound function, similar to the one introduced by Baruah. These concepts can be combined with event streams as shown in [2] and there is an approximation [2] and a fast analysis [1] available, too.

The demand bound function calculates for an interval of a given length the worst case amount of processor time which might be needed by intervals of these length to guarantee all deadlines within this interval. The idea for the feasibility analysis is to test for each possible interval if the processor capacity which is at least available within intervals of this length is sufficient to satisfy the demand for this interval. For simplicity the capacity is usually considered to have a constant growth and as the demand is measured in computation time the test can simply compare the demand with the length of the interval.

Def. 9: Request Bound Function: *The request bound function gives the maximum amount of computation time which is requested by a task set in any interval of length I.*

$$Rbf(I, \Gamma) = \sum_{\tau \in \Gamma} EBF(I, \Theta_\tau') \cdot C_\tau$$

Def. 10: Demand Bound Function: *The demand bound function describes the amount of requested computation time for a given time interval:*

$$Dbf(I, \Gamma) = \sum_{\tau \in \Gamma} EBF(I - d_\tau, \Theta_\tau') \cdot C_\tau$$

This demand bound function can be used to define a feasibility test for real-time systems with dynamic priorities, such as earliest deadline first scheduling (EDF), and scheduling with fixed priorities like rate monotonic scheduling.

Theorem 1: Processor Demand Test: *A set of real-time tasks is feasible if and only if*

$$\forall I \in N_0 \qquad Dbf(I, \Gamma) \leq I$$

A task set is feasible if the above condition holds for all intervals. It is only necessary to test special well known intervals (those for which the *demand bound function* can change) and it is possible to calculate an upper test bound for these intervals. See [2] and [3] for more details on the test and [1] for efficient algorithms to perform it.

For static priorities it is necessary to find an interval which is smaller than the deadline and in which the sum of the demand bound function of the actual task and the request bound functions of all tasks with a higher priority is equal or smaller than the length of the interval. See [3] for details.

6. Event Dependency Analysis

The purpose of the event dependency analysis is to quantify the event activity on the nodes of the event dependency graph. We describe this activity using the hierarchical event stream model. A methodology is needed to extract for a given task and an input event stream the outgoing event stream of the task. Different kinds of events lead to different event streams. The idea behind this analysis is to abstract from the internal control flow of the task. In contrast to the *SPI* model which allows the description of different output rates depending on the internal state of the task, the event dependency analysis delivers the worst-case behavior of the task for all possible execution paths of the task.

In our model the task consists of a control flow graph that may include branches and loops. Events can be generated at particular nodes of the graph. These nodes can for example be activations of other tasks or accesses to common memory. The purpose of event dependency analysis is to extract timing behavior of these events taking the different path of the control flow graph into account.

In the event dependency analysis the following steps have to be performed:

1. Traverse the graph step by step
2. Update at each step the event stream and additional event sequences necessary for the further calculation of the event stream. These event sequences will be introduced in the following chapter.

Let us consider graphs without loops first. Figure 3 illustrates the traversal through the graph. It starts at the node triggered by the external event stream. The graph is traversed node by node. At each step the successor of the last node is concatenated to the sub-graph visited so far. For nodes with more than one successor (due to if-statements) each following branch is considered separately. A union of the results is done at the nodes where the branches flow together again. Remaining branches are unified at the completion of the traversal.

The idea of the event dependency analysis is to calculate and update for each node the minimum event stream for all traversed nodes. At the end, when all nodes have been traversed, this minimum event stream is the resulting event stream for the whole task.

As basic operations we need methods to extract the event stream when (1) concatenating a node to a previously evaluated part of the graph (concatenation operation) or (2) unifying two different branches of the graph (merge operation).

6.1. Sequences

For these operations we need to define some specific sequences. There are four types of sequences to consider: The *start sequence*, the *end sequence*, the *inner sequence* and the *total sequence*. Note that for the different operations only a subset of all sequences are needed.

Def. 11: Start Sequence (StS): *The start sequence is a sequence were all intervals start with the beginning of the first node.*

Def. 12: End Sequence (ES): *The end sequence is a sequence where all intervals have the end of the graph as the common end point.*

Def. 13: Inner Sequence (IS): *The inner sequence includes for all possible numbers of events the shortest intervals that can be found somewhere in the graph and include at least these number of events.*

This inner sequence is also the resulting event stream.

Def. 14: Total Sequence (TS): *A total sequence is a sequence where all intervals have the start of the first node as common start point and the end of the last node as common end point. Each interval can represent another path through the graph. The different paths can generate different numbers of events and lead to different length of intervals. The total sequence includes for all possible*

number of events the shortest path through the complete graph which includes at least this number of events.

A problem is that we can have a path through the graph which does not generate any events. We need to represent this path too.

So we denote the total sequence as a tuple consisting on one side of the minimum interval α needed to traverse the graph and an event sequence describing the minimum intervals needed to traverse the graph and generate n events. $TS = (\alpha,\Theta)$

The total sequence is especially interesting for loop constructions. For its calculation not only the maximum number of iterations is taken into consideration but also each number of iterations less than the maximum number. Of course it is possible to define also a minimum number of iterations. In this case the length of each of the inner sequences has to cover at least this minimum number of iterations.

6.2. Initial values

The smallest possible unit that can be characterized by these sequences is a single node. It is then possible to build up entire graphs out of single nodes. Each node of the control flow graph is itself a graph and can be characterized by an initial start, end, inner and total sequence. We have two sets of initial sequences depending on whether the node generates an event or not.

A node with a best case execution time b generating an event has the following sequence:

$$StS = \left\{\begin{pmatrix} \infty & 1 \\ b & e \end{pmatrix}\right\} \qquad ES = \left\{\begin{pmatrix} \infty & 1 \\ 0 & e \end{pmatrix}\right\}$$

$$IS = \left\{\begin{pmatrix} \infty & 1 \\ 0 & e \end{pmatrix}\right\} \qquad TS = \left(b, \left\{\begin{pmatrix} \infty & 1 \\ b & e \end{pmatrix}\right\}\right)$$

For a node which does not generate an event the sequences looks as follows:

$$StS = \{\ \} \quad ES = \{\ \}$$
$$IS = \{\ \} \quad TS = (b,\ \{\ \})$$

Note that we consider (without loss of generality) events to be generated at the end of the nodes.

6.3. Operators

To build up the sequences for complex graphs out of the initial sequences two basic operations are needed, the merge and the concatenation operation.

Def. 15: Merge Operation: *A sequence Θ_A is the merged sequence of two sequences Θ_B and Θ_C if for all intervals I: $EBF(I, \Theta_A) = max(EBF(I, \Theta_B), EBF(I, \Theta_C))$*

With the merge operation the maximum number of events of the two seqeunces for any interval can be calculated. If the sequences are valid event streams the resulting sequence will also be a valid event stream.

Def. 16: Concatenation Operation: *A sequence Θ_A is the concatenation of two sequences Θ_B and Θ_C if for all intervals I_A, I_B, I_C the event bound function $EBF(I_A, \Theta_A)$ is equal to the maximum sum of the event bound functions $EBF(I_B, \Theta_B)$ and $EBF(I_C, \Theta_C)$ for all possible combinations of I_B and I_C fullfilling $I_A = I_B + I_C$.*

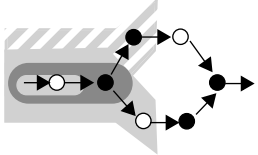


Figure 3: Principles of the event dependency analysis

The implementation of these operations will be discussed in Section 7.1 and Section 7.2.

6.4. Evaluation of simple task graphs

First the simple case that the task is triggered only once will be considered. Every traversal step is a merge operation between a previously traversed part of the graph and the following node. As the node is itself a graph, the merge can be regarded as a merge operation between two graphs. The problem is therefore reduced to the problem to calculate the sequences of the merged graph using only the sequences of the two parts of the graph.

Let us consider each kind of sequence separately. The different possible paths resulting out of the merge can lead to intervals of different length for the same number of events. The resulting sequence includes for each possible number of events the minimum of these intervals leading to the maximum sequence.

Let us consider the concatenation of two graphs *A* and *B* to a resulting graph *C*. The intervals for the start sequence of *C* can either come from the start sequence of *A* or from paths which include *A* completely. These paths are the result of a concatenation between the total sequence of *A* and the start sequence of *B*.

For the end sequence of *C* we need a union of the end sequence of *B* and the concatenation between the inner sequence of *B* and the end sequence of *A*.

The inner sequence of *C* can be generated by a union of the inner sequence of *A* and *B* and an additional union of the result with the concatenation between the end sequence of *A* and the start sequence of *B*.

The new total sequence of *C* results from a concatenation of the total sequences of *A* and *B*.

For the union of branches the resulting start, end, inner and total sequences are calculated by a union between the corresponding sequences of the parts.

The resulting event stream is the inner sequence. For calculating the inner sequence only the inner and the end sequence of *A* are needed, the start and the total sequence are of no relevance. Therefore it is only necessary to keep track of the end and inner sequences during the traversal of a graph.

Lemma 5: *The resulting inner sequence is a valid event stream. This event stream represents the worst case density of events in any possible path of the graph.*

Proof: This lemma is obviously true for single nodes. It is necessary to show that the previous operations preserves this condition. The proof is done by induction. Let us first consider the merge of two graphs. The worst case density for a certain number of events in the resulting inner sequence has to be included in one of the previous inner sequences. Due to the definition of the merge operation it therefore remains in the resulting inner sequence. For the start, end and total sequence the condition is the same. For the concatenation the resulting worst case density for a certain number of events in the inner sequence has to exist either in one of the previous inner sequences or has to include the connection point between the previous sequences. All these cases are merged, so the resulting inner sequence includes the worst case of all possibilities. The proof for the start and end sequence follows

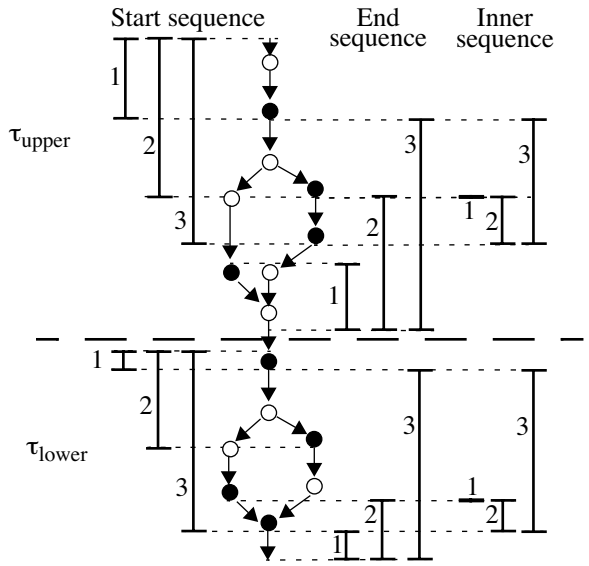


Figure 4: Sequences of the event dependency analysis accordingly. □

7. Concepts for Implementation

The implementation that we introduce next is not limited to homogenous sequences. Instead it is advisable to introduce a weaker condition.

Def. 17: Strict Order Condition: *A hierarchical sequence is in strict order if in any period the event of an element occurs before the event of another element in the same period, its events must in all allowed periods occur before the corresponding events of the other element.*

The allowed periods are given by the limitation of the hierarchical event sequence elements. A homogeneous event sequence is always in strict order. An non-homogeneous event sequence can only be in strict order if the number of generated events is limited.

7.1. Merge Operation

In the following we will discuss the principles of implementation for the merging operation of two (or more) event sequences. The resulting (merged) event sequence includes for all intervals the maximum number of events which can occur in one of the original sequences. If they are valid event streams the resulting hierarchical event sequence will also be a valid event stream.

Def. 18: Domination: *An event sequence* Θ_1 *dominates an event sequence* Θ_2 *for a specific number of events n if*

$$IF(n, \Theta_1) \leq IF(n, \Theta_2)$$

It is necessary to split the complete range of numbers of events into parts which have a clear unique domination of a sequence. We call these parts domination ranges.

In Figure 5 the process of determining the domination for two simple event sequence elements is shown. Remember that a simple element is characterized by a period and an initial interval. The event bound function of each element is a straight line with the initial interval as start value and the period as gradient. Comparing two simple event sequence elements leads to two possible scenarios. Either one element dominates over the complete range, or the domination changes. This can happen if one sequence has

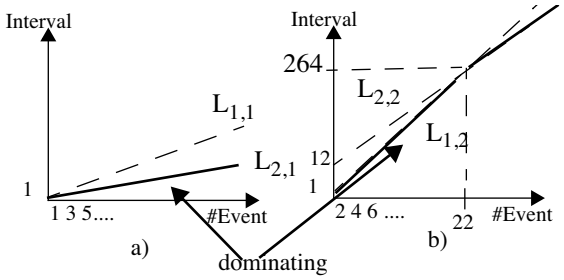


Figure 5: Merging the HES for two loops.
a) First event elements; b) second event elements of L_1, L_2

a smaller initial interval but a larger period than the other sequence. Then this sequence will dominate first until the intersection of the two lines. There the domination changes. For two sequence elements ω_1, ω_2 the intersection x is: $x = (a_1 - a_2)/(p_2 - p_1)$

In the first case the dominating element is added to the resulting event sequence. In the second case two elements are added. The first dominating element is added with a limitation that covers all events up to the intersection. Next the second dominating element is added with the intersection as additional offset.

For an event sequence consisting of several elements the domination can change frequently. To re-establish the behavior it is necessary to split the sequence into single elements. The idea behind the merging operation is to compare those elements of both sequences which lead to the same numbers of events. After sorting, elements on the same position can be compared with each other. It is necessary to expand the contributing sequences to the same number of elements. This is the least common multiple (LCM) of the number of the elements of all contributing sequences. Note that the periods are of no concern here.

For hierarchical sequence elements the length of the straight line is limited by the number of events possible by this element.

We will now give a little example. Consider Figure 6. It shows two control flow graphs each consisting of a loop. The event sequence representing these loops can be written as follows:

$$\Theta_3 = \left\{ \begin{pmatrix} 25 & 1 \\ 0 & e \end{pmatrix}, \begin{pmatrix} 25 & 1 \\ 1 & e \end{pmatrix} \right\} \qquad \Theta_4 = \left\{ \begin{pmatrix} 12 & 1 \\ 0 & e \end{pmatrix} \right\}$$

$$\Theta_1 = \left\{ \begin{pmatrix} \infty, 200 \\ 0, \Theta_3 \end{pmatrix} \right\} \qquad \Theta_2 = \left\{ \begin{pmatrix} \infty, 100 \\ 0, \Theta_4 \end{pmatrix} \right\}$$

To merge these streams Θ_4 has to be expanded to two

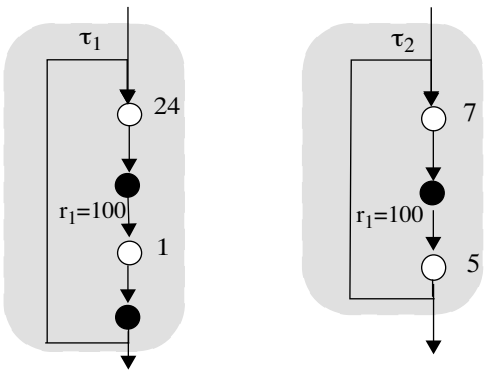


Figure 6: Merging task graphs of two loops

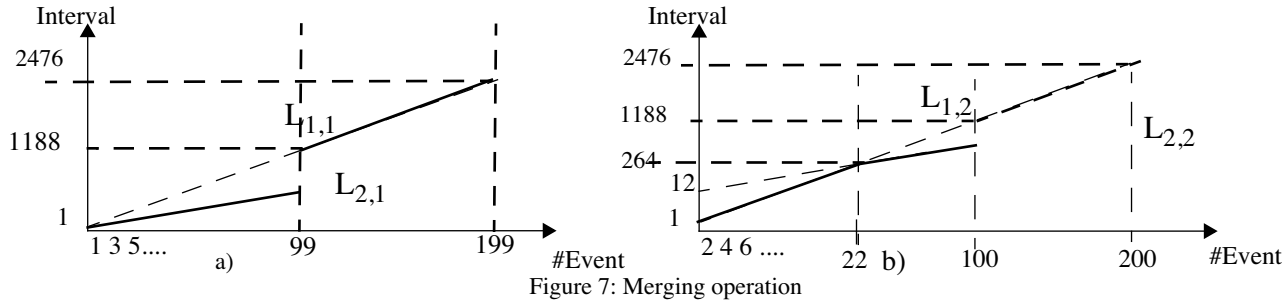


Figure 7: Merging operation

a) First event elements; b) second event elements .Step-by-step linear merge and finding the dominating elements. For a) there are two pieces, $L_{2,1}$up to 99 events and $L_{1,1}$ for the rest. In b) there are three pieces

event stream elements using a new period of 24:

$$\Theta_1 = \left\{\binom{25}{0}, \binom{25}{1}\right\} \qquad \Theta_2 = \left\{\binom{24}{0}, \binom{24}{12}\right\}$$

Than the event stream elements of both streams can be compared pairwise. The first element of each stream is responsible for the intervals containing at most 1,3,5,7,9,... events, the second element is responsible for the intervals containing 2,4,6,8,... elements. Figure 7 shows the domination graph for the example. Next the domination intervals can be calculated. The resulting event sequence is:

$$\Theta_r = \left\{\binom{\infty, 50}{0, \binom{24}{0}}, \binom{\infty, 11}{0, \binom{25}{1}}, \binom{\infty, 39}{264, \binom{24}{12}}, \binom{\infty, 100}{1188, \Theta_3}\right\}$$

The same sequence can be also rewritten as:

$$\Theta_r = \left\{\left(\begin{matrix}\infty, 22\\ 0, \left\{\binom{24}{0}, \binom{25}{1}\right\}\end{matrix}\right), \left(\begin{matrix}\infty, 78\\ 264, \left\{\binom{12}{0}\right\}\end{matrix}\right), \binom{\infty, 100}{1188, \Theta_3}\right\}$$

The resulting sequences can be extracted step by step. The original sequences are compared step by step to find for each segment the domination elements and to merge these elements to the resulting sequence. In the example above, the first segment ends at the event with the number 100. In this segment for the respective first elements of the contributing sequences Θ_1 dominates within the complete segment. This leads to one resulting element. For the respective second elements the domination changes within the segment. This results in two elements, one for the first part of the segment up to the intersection, one for the second part. The second segment runs from event 101 to event 200 and in it only Θ_1 can contribute which results in one element in the resulting sequence for each of the elements of Θ_1 .

7.2. Concatenation operation

The concatenation of the end sequence of a graph with the start sequence of the following graph is done to find out if this leads to new more strict elements for the inner sequence. For the merge operation each element in a sequence could generate the intervals for specific numbers of events. For the concatenation operation the contribution scheme is different. Each element can generate intervals for each possible number of events which is larger than the initial number of events of the element.

The reason is that the intervals can be completed to the necessary number of events by a corresponding element of the other sequence. Consider the following example which is illustrated in Figure 8:

$$\Theta_1 = \begin{pmatrix} \infty & 200 \\ 0 & \left\{\binom{25}{0}, \binom{25}{1}\right\} \end{pmatrix} \qquad \Theta_2 = \begin{pmatrix} \infty & 100 \\ 0 & \left\{\binom{24}{7}, \binom{24}{19}\right\} \end{pmatrix}$$

Θ_1 is an end sequence and Θ_2 is a start sequence. Let us first consider the embedded sequences alone and than introduce the limitation. For one event it is not possible to find an interval including the connection point between the two graphs, therefore the initial interval is added to the resulting sequence. For two events there is only one possibility, the concatenation of the first elements of both sequences. This leads in this example to an interval of size 7. So we have

$$\binom{\infty}{0}, \binom{\infty}{7}$$

as contribution for the aperiodic part. For 3,5,7,9,... events either combinations of the second element of the first sequence ($\omega_{1,2}$) and the first element of the second sequence ($\omega_{2,1}$) or the combination of $\omega_{1,1}$ and $\omega_{2,2}$ are possible. For 5,7,9... events these combinations can be extended periodically by the periods of one or both of the elements involved in the combination. For the dominating intervals only the smaller period in a combination is important because both periods start with the same offset. This offset is the distance between the first events of both elements. For four events the combination $\omega_{1,1}$ and $\omega_{2,1}$ (extended by one period) and the combination $\omega_{1,2}$ and $\omega_{2,2}$ are possible. They can be extended using the periods of the elements to 6,8,10,12,... events. Again, the element with the shorter period dominates. Some combinations never dominate. These combinations can be found and removed by comparing the initial distances between the elements and the periods with each other element, as shown in Figure 9. For 3,5,7,9,... events we have the following possible combinations:

$$\binom{25}{8}, \binom{24}{19}, \binom{25}{19}, \binom{24}{8}$$

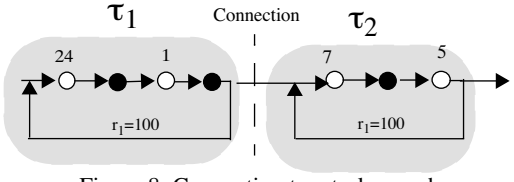
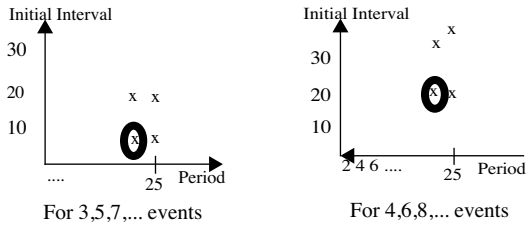


Figure 8: Connecting two tasks graphs



For 3,5,7,... events For 4,6,8,... events

Figure 9: Comparing possible combinations

For 4,6,8,10,... events these combinations are possible:

$$\binom{25}{32}, \binom{24}{31}, \binom{25}{20}, \binom{24}{20}$$

Only Pareto-optimal points can dominate for some time. These are all points having no point better in one condition and at least equal in the others. So for 3,5,7,9... events we have the following initial sequence:

$$\left\{\binom{24}{8}, \binom{24}{20}\right\}$$

And for 4,6,8... events:

$$\left\{\binom{12}{8}\right\}$$

This would be sufficient for simple event sequences with infinite periods. For hierarchical event sequences the limitation of the number of events has to be considered. For a combination the element with the smaller period is used first to generate events. When reaching the limitation for this element, the possible following domination can be either a following element in the sequence or the partner element of the other sequence.

Taking all this into account the event seqeunce for the example is:

$$\left\{\binom{\infty}{0}, \binom{\infty}{7}, \begin{pmatrix} \infty & 98 \\ 8 & \left\{\binom{12}{0}\right\} \end{pmatrix}, \begin{pmatrix} \infty & 200 \\ 1220 & \left\{\binom{25}{0}, \binom{25}{1}\right\} \end{pmatrix}\right\}$$

8. Loops

A simple approach is to unroll the loops and consider the chain of nodes with additional edges modelling the conditional exit of the loop before the maximum number of allowed iterations is reached. However with loop-unrolling the number of iterations of the loop affects the complexity of the analysis. To efficiently evaluate a loop we first calculate the necessary sequences for the inner part of the loop. This can be done using the methodology introduced in the previous chapters. After that we have to combine the sequences with the maximum and minimum allowed iterations. This can be done by a continuous concatenation of the inner parts of the loop. So for five iterations we have to concatenate the inner part five times. Note that for the resulting total sequence we have to merge each allowed number of iterations, that means each number between the minimum and the maximum number of iterations.

Because the same graphs are concatenated repeatedly it is evident that the resulting event sequences will have recurring patterns. These patterns can elegantly be described using hierarchical event sequences. Therefore it is possible to extract these resulting hierarchical sequences directly out of the structure of the loop.

9. Complexity

We have to differentiate between the complexity of the analysis and the complexity to achieve the hierarchical event streams. The real-time analysis itself has pseudo-polynomial complexity in the number of elements. Adapting the approximations proposed in [1],[2],[11] would lead to a polynomial-time approximation for both static and dynamic priority systems.

The algorithm to extract the event streams generally has a polynomial complexity ($O(n^2)$) with regard to the number of nodes that generate events. The algorithm leads to a linear grows of the number of elements in the resulting event stream at most. However it can have exponential complexity with regard to the number of those loops which can generate events. This complexity is bound by the cumulated number of iterations of these loops. It is important that the complexity of the model and the approximation analysis is independent of the number of loop iterations.

10. Case Study

In the following we will show the advantages of the new model using a well known case study.

The Synthetic Aperture Radar application is used to create high resolution pictures even under adverse visiability conditions. The benchmark is the standard simplified version of the original system where the size of data packages has been reduced.

To show the possibilities and advantages of the new model we will consider the system implementation shown in Fig 12. We consider the system to run nearly completely on one processor, only the calculation for the fast fourier transformation are considered to be implemented on a specialized co-processor. As we are interested in the load on the co-processor we need the triggering event stream for it. We have transferred the original C-code of the benchmark into control-flow-graphs and then into an event dependency graph. We used *ChronEst* [18] to extract execution times for the single nodes of the graphs.

The resulting hierarchical event stream looks as follows:

$$\Theta_1 = \begin{Bmatrix} \infty & 40960 \\ 0 & \Theta_2 \end{Bmatrix}$$

$$\Theta_2 = \left\{\binom{6477}{0}, \begin{pmatrix} 6477 & 512 \\ 4.9 & \left\{\binom{9.58}{0}\right\} \end{pmatrix}, \begin{pmatrix} 6477 & 64 \\ 4898.6 & \left\{\binom{12.42}{0}\right\} \end{pmatrix}, \begin{pmatrix} 6477 & 63 \\ 5706 & \left\{\binom{12.42}{0}\right\} \end{pmatrix}\right\}$$

It can be seen that the stimuli are generated within a nested loop. The first two events have a distance of 4.9 time units followed by three bursts. The first burst consist of 512 additional events each having a distance of 9.58 to the previous event. It is followed by two bursts with 64 and 63 events, both having a period for the inner events of 12.42. The distances between the three bursts are different.

In the following we show our attempts to model the same behavior with previously existing models. Our focus was

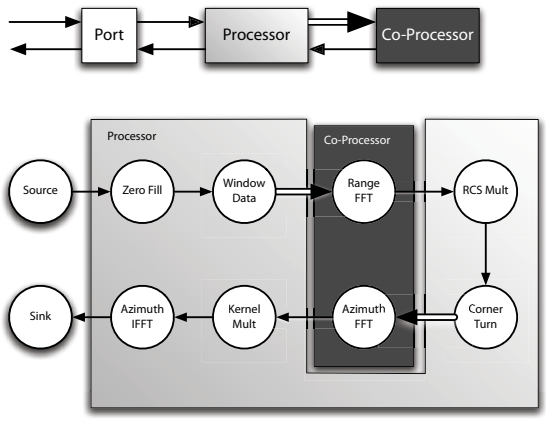


Figure 10: Used Hardware Binding for the SAR Application

to model the stream exactly, e.g. without the loss of information and accuracy.

In the previous event stream model it would be necessary to add one event sequence element for each of the 2560 events. This would greatly increase the run-time of the analysis as it is determined by the number of elements. Even if the outer loop would continue infinitely, each element of the burst would have to be modeled by its own element which would lead to an event stream with 640 tuples.

But even considering a model specifically designed to cover bursts does not improve the situation. In the model of Chakraborty and Thiele [7] the stimuli is represented as a set of tuples (α,Δ) where α is the maximum number of events that can occur in an interval of length Δ. An interval of size 2Δ, can than contain a maximum of 2α events and so on. Each interval is limited by all tuples and for the analysis it is necessary to calculate all possible linear combinations of the tuples. This leads to exponential complexity of the analysis and would also include a complete unrolling of the loops, an additional complexity prevented in our approach.

A first attempt to model the stimuli can be as follows:

$$\{(1,4.9), (511,4896.6), (575,5706), (637,6477), (2560,\infty)\}$$

But this description of the stimuli is too pessimistic. It allows the first 512 events to occur with a distance of 4.9 to their successor. Just as in the other bursts the internal period would be 4.9. To capture the behavior exactly it is necessary to add a tuple for each element within the burst, just as in the classical event stream model. Even a possible extension of this model with offsets would not be satisfactory due to the analysis complexity of the model. Additionally such an extension would come close to our proposed model, for which an efficient analysis is available. The new model combines both bursts and periodic / aperiodic behavior and captures complicated systems accurately and efficiently.

11.Conclusion

In this paper we have introduced a new complete methodology, called event dependency analysis, for the analysis of event driven distributed real-time systems covering the path from source-code level to the real-time analysis. This methodology allows to extract accurately the timing of dynamic task activations. These are activations of tasks generated by other tasks anytime during their execution.

We proposed a method to extract the timing of theses dynamic task activations out of the control flow graphs of the system.

We have introduced a new model for the description of the timing behavior of events which is able for the first time to efficiently describe both bursts and aperiodic behavior which in turn allows an efficient description of events generated within loops. It is well suited not only for the event dependency analysis. We provided an efficient feasibility analysis for real time systems based on the concepts of the demand and request bound function. The quantitative description of the model allows the extension to approximative feasibility analysis which we have postponed to future work.

We validated the suitability of our approach using the SAR benchmark. We especially showed that the hierarchical event streams can model situations accurately and efficiently which cannot be captured with the same precision by competing models.

12.References

[1] K. Albers, F. Slomka. *Efficient Feasibility Analysis for Real-Time Systems with EDF Scheduling*. IEEE Proceedings of the Design Automation and Test in Europe Conference (DATE'05), pp. 492-497, 2005.

[2] K. Albers, F.Slomka. *An Event Stream Driven Approximation for the Analysis of Real-Time Systems*, IEEE Proceedings of the 16th Euromicro Conference on Real-Time Systems, pp. 187-195, 2004.

[3] S. Baruah. *Dynamic and static-priority scheduling of recurring real-time tasks*. Real-Time Systems, 24(1) pp. 93-128, 2003

[4] S. Baruah, D. Chen, S. Gorinsky, A. Mok. *Generalized Multiframe Tasks*. The International Journal of Time-Critical Computing Systems, 17, 5-22, 1999.

[5] S. Baruah, A. Mok, L. Rosier. *Preemptive Scheduling Hard-Real-Time Sporadic Tasks on One Processor*. Proceedings of the Real-Time Systems Symposium, 182-190, 1990.

[6] J.Y. Le Boudec, P. Thiran. *Network Calculus - A Theory of deterministic Queuing Systems for the Internet*. LNCS 2050, Springer Verlag, 2001.

[7] S. Chakraborty, L. Thiele. *A New Task Model for Streaming Applications and its Schedulability Analysis*. IEEE Proceedings of the Design Automation and Test in Europe Conference (DATE'05), pp. 486-491, 2005

[8] S. Chakraborty, S. Künzli, L. Thiele. A. Herkersdorf, P. Sagmeister. *Performace Evaluation of Network Processor Architectures: Combining Simulation with Analytical Estimation*, Computer Networks, Vol. 41, No. 5, pp. 641-665, 2003.

[9] S. Chakraborty, S. Künzli, L. Thiele. *Approximate Schedulability Analysis*. 23rd IEEE Real-Time Systems Symposium (RTSS), IEEE Press, 159-168, 2002.

[10] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich. *Hardware/software codesign of embedded systems - the spi workbench*. In Proceedings of the IEEE Computer Society Workshop on VLSI'99, 1999.

[11] N. Fisher, S. Baruah *A polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines*. Proceeding of the 17th Euromicro Conference on Real-Time Systems, Palma de Mallorca, Spain. July 2005

[12] A. Maxiaguine, S. Künzli, L. Thiele. *Workload Characterization for Tasks with Variable Execution Demand*. IEEE Proceedings of the Design Automation and Test in Europe Conference (DATE'04), pp. 1040-1045, 2004

[13] M. Jersak, R. Henia, R.Ernst. *Context-Aware Performance Analysis for Efficient Embedded System Design*. IEEE Proceedings of the Design Automation and Test in Europe Conference (DATE'04), pp. 1046-1051, 2004

[14] S. Goddard, X. Liu. *A Variable Rate Execution Model*. Proceedings of the 16th Euromicro Conference on Real-Time Systems, 2004

[15] K. Gresser. *An event model for deadline verification of hard real-time systems*. In 5th Euromicro Workshop on Real-Time Systems, Finland, 1993.

[16] K. Jeffay, S. Goddard. *A Theory of Rate-Based Execution*. Proceedings of the 20th IEEE Real-Time Systems Symposium, pp. 304-314, 1999

[17] C. Liu, J. Layland. *Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments*. Journal of the ACM, 20(1), 46-61, 1973.

[18] F. Slomka. *New Techniques for the Design of Distributed Embedded Real-Time Systems*. Proceedings of the Embedded World Conference. Nürnberg, Germany. February 2005.