Interface-Based Design of Real-Time Systems with Hierarchical Scheduling

Ernesto Wandeler Lothar Thiele
Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich, Switzerland
E-mail: {wandeler,thiele}@tik.ee.ethz.ch

Abstract— In interface-based design, components are described by a component interface. In contrast to a component description that describes what a component does, a component interface describes how a component can be used, and a well designed component interface provides enough information to decide whether two or more components can work together properly in a system. Real-Time Interfaces expand the idea of interface-based design to the area of real-time system design, where the term of working together properly refers to questions like: Does the composed system satisfy all requested real-time properties such as delay and throughput constraints? In this work, we extend the theory of Real-Time Interfaces and prove its applicability for the design of systems with hierarchical scheduling. We introduce a component system for interface-based design of systems with mixed FP, RM and EDF scheduling. We then further extend the ability for hierarchic scheduling by introducing server components. The introduced component system with Real-Time Interfaces not only allows interface-based design of complex real-time systems with hierarchical scheduling, but also inherently enables detailed schedulability analysis of such systems.

1 Introduction

The increasing application and system complexity of modern real-time embedded systems demands for new compositional design and analysis methods. These need to be able to decide whether sets of individual system components can work together properly in a system design, especially in terms of non-functional constraints, such as buffer space, delays and throughput. If done properly, we can expect that with such methods, future real-time system design could benefit from the properties of incremental design and independent implementability, that are the keys for designing complex systems with interacting components.

An important requirement for compositional real-time system design and analysis is the isolation of the non-functional behavior of different system components, as it is partly achieved for example by means of hierarchical scheduling methods, see e.g. [12]. This way, changes in certain aspects of the implementation of single system components do not interfere with the rest of the system in any unexpected manner.

The compositional design and analysis approach we are proposing in this paper is based on the theory of Real-Time Interfaces [15] that combines Interface-Based Design as in [7] with the analysis of real-time systems using Real-Time Calculus as in [6]. Real-Time Interfaces enable interface-based design of real-time systems, where compliance to non-functional constraints of a system is checked at composition time. This leads to faster design processes and partly removes the need for the classical trial-and-error approach to find economically dimensioned systems. And because of the inherent constraint propagation, the use of Real-Time Interfaces also allows instantaneous answers to typical design questions like: If I put these components together, what resources remain for additional components? What is the maximal task activation rate I can support? What is the minimal task execution delay I can ask for? What is the least hardware resource I need, to guarantee all timing constraints?

This approach is in contrast to classical approaches where functional component behavior is the major focus, see for example the use of Corba/Java components in [16].

A first step towards the proposed direction has been described in the context of hierarchical scheduling analysis, see e.g. [12, 13, 2, 4]. Common to all of this previous work is the use of very restricted models to propagate constraints (resource demands) between different hierarchy layers. All methods use either a periodic or a bounded delay resource model to model the resource demand at a given level of hierarchy. The actual resource demand at a given level of hierarchy is however typically more complex, and as a consequence, a considerable abstraction overhead is introduced at every level of hierarchy. To show an example, the lower curves in the graphs in Fig. 1 depict the actual resource demand per time interval of four tasks that are scheduled using rate monotonic scheduling (tasks $T7 - T10$ from Table 1 are used here). If we model this resource demand using a periodic resource model (left) or a bounded delay resource model (right), a considerable abstraction overhead is introduced by the use of these coarse grained resource models.

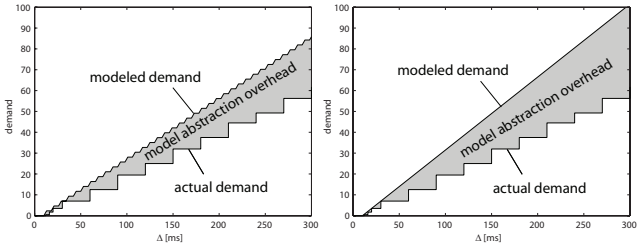

Figure 1. Overhead in previous work.

Moreover, the methods presented in [12, 13, 4] only consider systems that are triggered by strictly periodic event streams, while [2] considers periodic event streams with jitter.

In contrast to this previous work, the framework presented in this work makes use of a fine-grained resource model that exactly captures and propagates the required resource demand at any given level of hierarchy (the exact resource demand in Fig. 1 is captured by $\hat{\beta}^A_{RM}$ in Fig.10). Further, the use of a likewise fine-grained event stream model enables the presented framework to analyze systems with arbitrary complex event stream patterns, including bursty event streams. And finally, the presented framework is based on the sound theories of Interface-based Design and Real-Time Calculus, and can hence fall back to a wide range of already established research results in these areas.

Contributions of this work:

- We present a component system for Interface-Based Design of real-time systems with mixed FP, RM and EDF scheduling.
- We then extend the ability for hierarchic scheduling in this component system by introducing server components. As a result, arbitrary hierarchies of scheduling policies can be dealt with.
- An extensive example is provided that shows the capabilities of the approach in terms of hierarchical composition, combination of different event models and scheduling strategies, detailed real-time analysis of complex systems, and answering design questions (maximal event rates, minimal deadlines, minimal processing capabilities) through constraint propagation. The example also reveals that the methods can be implemented computationally efficient.

2 Interface-Based Design

The definition of Real-Time Interfaces follows the principles of Interface-based Design as described by de Alfaro and Henzinger in [7] and more recently [9]. Whereas most previous results relate to stateful interface languages such as interface automata [8] and extensions towards the use of resources [5], the Real-Time Interfaces are based on stateless assume/guarantee (A/G) interfaces, see [7]. In this section, we introduce some underlying principles of interfaces and Interface-based Design on an example of an simple imaginary component for real-time system design. Note that interfaces and all interface-related terms in this section are formally defined in [7] and [9].

Imagine the simple component depicted in Fig. 2 (a). The component has two *input variables* a and b, and one *output variable* c. What the component does is further described by a *component description* that is expressed by the formula $c = b - a$. To put this component in a context, lets suppose that it is the abstraction of a concrete system component, that is used to analyze real-time properties of the concrete component. We could then interpret this component as follows: input variable a describes the resource demand of an arriving event stream, while b describes the resources that are available to process the arriving event stream. Output variable c would then describe the resources that remained unused after processing the resource demand a.

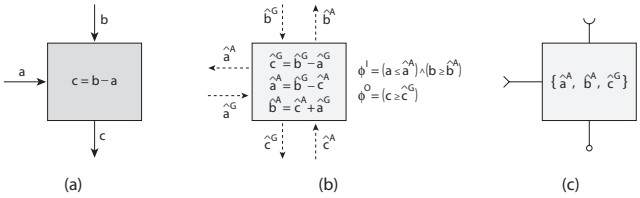

Figure 2. A component (a), its interface (b), and its interface block diagram (c).

In contrast to the component description, that describes what the component does, the *component interface* describes how the component can be used. And a well designed component interface provides enough information to decide whether two or more components can work together properly in a system.

Figure 2 (b) depicts an assume/guarantee interface for the component in Fig. 2 (a). Although not depicted explicitly, this interface also has the two *input variables* a and b and the *output variable* c. The component interface puts a constraint on the environment through a predicate ϕ^I on its input variables: the environment is expected to provide inputs that satisfy ϕ^I. In return, the interface communicates to the environment a constraint ϕ^O on its output variables: it guarantees to provide only outputs that satisfy ϕ^O.

If our interface gets connected to other interfaces, these will provide output guarantees on their output variables, e.g. $a \leq \hat{a}^G$ and $b \geq \hat{b}^G$, and input assumptions on their input variables, e.g. $c \geq \hat{c}^A$. The *interface relations* $\hat{c}^G = \hat{b}^G - \hat{a}^G$, $\hat{a}^A = \hat{b}^G - \hat{c}^A$ and $\hat{b}^A = \hat{a}^G + \hat{c}^A$, then bring these different input assumptions and output guarantees into relation.

For interface-based design, we first need a set of interfaces, one for every component in the system we want to design. As with the components themselves, interfaces can then be composed into bigger interfaces by interconnecting an output of one interface U to an input of another interface V. The composed interface is then denoted as $U||V$. This composition is however only possible, if the two interfaces are semantically *compatible*. And if they are compatible, it is guaranteed that the two components belonging to the two interfaces can work together properly in the real system.

Two interfaces U and V are semantically compatible if whenever one interface provides inputs to the other interface, then the output guarantee of the former implies the input assumption of the latter. In the closed case, where all inputs of U are outputs of V, and vice versa, U and V are compatible if the closed formula $\phi_U^O \wedge \phi_V^O \Rightarrow \phi_U^I \wedge \phi_V^I$ is true. In the open case on the other hand, where some inputs of U and V are left free, this formula has free input variables. U and V are then compatible if the above formula is satisfiable. This formula is then the input assumption $\phi_{U||V}^I$ of the composite interface $U||V$, as it encodes the weakest condition on the environment of $U||V$ to make U and V work together properly.

Interface-based design supports *incremental design* of systems, because interfaces can be composed one-by-one in any order. During the composition, the assumptions on the free input variables are getting increasingly tight, and if we eventually succeed to compose all component interfaces of a complete system, we are guaranteed that all components in the system work together properly.

Besides composition, the second operation on interfaces is *refinement*. Refinement of interfaces is very similar to subtyping of classes in OO programming: a refinement of an interface must accept all inputs that the original interface accepts, and it may produce only outputs that the original interface specification allows. Hence, to refine an assume/guarantee interface, the input assumption can be weakened, and the output guarantee can be strengthened. This definition ensures that compatible interfaces can always be refined independently and still remain compatible. We then say that interface-based design supports *independent implementability*. In practice, this allows to outsource the implementation of different system components, or to replace existing implementations of sub-systems with different or new implementations.

To conclude, let us note that one can intuitively think of Interface-based Design with assume/guarantee interfaces as a well-defined method for constraints propagation in a component system.

3 Real-Time Interfaces

Real-Time Interfaces as introduced in [15] can be considered as a special instance of assume/guarantee interfaces, tailored towards assumptions and guarantees on the throughput and delay of events and the availability of resources. Based on the example in the previous section, we identify three steps that eventually lead to a component system for Interface-based Design of real-time systems:

- First, we need an abstract component that describes the real-time properties of a concrete HW/SW system component. For this, we need proper abstractions for the different component *inputs* and *outputs* as well as a *component description*, that meaningfully relates the inputs of the component to its outputs.
- Secondly, to obtain the interface of such an abstract component, we need to define the *interface variables* as well as the input and output *predicates* on the interface variables.
- Finally, we need to establish *interface relations* for the interfaces of different abstract components.

The following sections 3.1, 3.2 and 3.3 are dedicated to step 1, while section 3.4 is dedicated to step 2. Step 3 builds the central part of this work, and sections 4 and 5 are dedicated to it.

3.1 From Components to Abstract Components

In a real-time system, an incoming event stream is typically processed on a HW/SW component that can be interpreted as a task that executes on a hardware resource. Every event stream has an associated maximum delay requirement (relative deadline), and if several HW/SW components execute on the same resource, a scheduling policy manages the resource sharing.

To abstractly model a HW/SW component, we use Real-Time Calculus [14, 6]. In comparison to a concrete HW/SW component, an abstract component processes an abstract event stream on an abstract resource. In Real-Time Calculus, the timing and resource demand properties of the event stream that triggers the HW/SW component are abstracted by a Variability Characterization Curve (VCC) that is called arrival curve $\alpha(\Delta)$ following [10]. Together, $\alpha(\Delta)$ and the maximum allowable delay d describe the properties of an event stream that are essential for real-time analysis, see [14]. The hardware resource that enables the execution of tasks is also modeled by a VCC, called service curve $\beta(\Delta)$.

3.2 Variability Characterization Curves

An arrival curve $\alpha(\Delta) \in \mathbb{R}^{\geq 0}$, $\Delta \in \mathbb{R}^{\geq 0}$ provides an upper bound on the resource demand that events arriving in *any* time interval of length Δ create on a component, i.e. the events arriving within a time interval $[t, t+\Delta)$ may create a resource demand of at most $\alpha(\Delta)$ for all $t \geq 0$.

Arrival curves substantially generalize the classical representation of standard event arrival patters such as sporadic, periodic, periodic with jitter, or others. A classical Liu & Layland event model $T(p, e)$ with period p and event execution demand e can for example be modeled by an arrival curve $\alpha(\Delta) = \lceil \Delta/p \rceil \cdot e$ and a maximum delay $d = p$. Besides being able to represent any event stream with deterministic timing behavior, it is also possible to determine arrival curves corresponding to any finite length event trace, obtained for example from observation or simulation. Figure 3 depicts some examples of arrival curves.

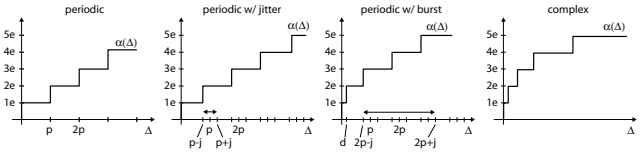

Figure 3. Examples of arrival curves.

Analogously, a service curve $\beta(\Delta) \in \mathbb{R}^{\geq 0}$, $\Delta \in \mathbb{R}^{\geq 0}$ provides a lower bound on the available resources in any time interval of length Δ, i.e. in any time interval of length Δ at least $\beta(\Delta)$ resource units are available.

Figure 3 shows the service curves of a resource that is fully available, a bounded delay resource $\Phi(\alpha, \Delta)$ [13], a periodic resource $\Gamma(\Pi, \Theta)$ [12], as well as of a resource with a more complex timing behavior.

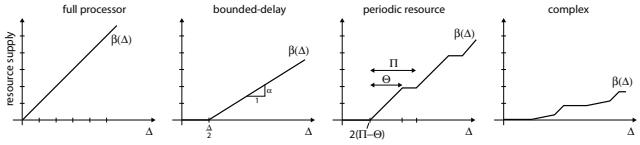

Figure 4. Examples of service curves.

3.3 Abstract Components

In this work, we model a HW/SW component using an abstract component as depicted in Fig. 5 (a). An event stream, represented by the arrival curve $\alpha(\Delta)$ with associated maximum delay d, triggers the component. A fully preemptable and independent tasks is instantiated at every event arrival and is processed greedily while being restricted by the resource availability, that is represented by the service curve $\beta(\Delta)$. Resources that are not consumed by the component are represented by the service curve $\beta'(\Delta)$.

Following the results from Network Calculus and Real-Time Calculus, the following relations can be derived that describe such an abstract component:

- **Remaining Service**: If an event stream with arrival curve $\alpha(\Delta)$ is processed by an abstract component on a resource with availability $\beta(\Delta)$, then the remaining resources that are not consumed by the abstract component can be bounded by the service curve:

$$\beta'(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{\beta(\lambda) - \alpha(\lambda)\} \stackrel{def}{=} RT(\beta, \alpha) \quad (1)$$

- **Delay**: The maximum delay d_{max} experienced by an event on an event stream with arrival curve $\alpha(\Delta)$ that is processed on an abstract component with service curve $\beta(\Delta)$ is bounded by:

$$\begin{aligned} d_{max} &\leq \sup_{\lambda \geq 0} \left\{ \inf\{\tau \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + \tau)\} \right\} \\ &\stackrel{def}{=} Del(\alpha^u, \beta^l) \end{aligned} \quad (2)$$

With abstract components as defined above, scheduling policies on a resource can be expressed by the way the abstract resources $\beta(\Delta)$ are distributed among the different abstract components. For example, consider preemptive fixed priority scheduling: an abstract component A with the highest priority may use all resources, whereas an abstract component B with the second highest priority only gets resources that were not consumed by A. This is modeled by using the remaining service $\beta'_A(\Delta)$ from A as input to B. For more details see [6].

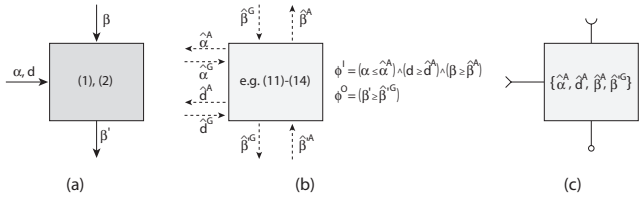

Figure 5. An abstract component (a), its Real-Time Interface (b), and its block diagram (c).

3.4 Real-Time Interface Variables and Predicates

A Real-Time Interface may have input and output variables related to event streams (*arrival variables*) and resource availability (*service variables*). An arrival variable consists of an arrival curve $\alpha(\Delta)$ and an associated maximum event delay d. The output guarantee on an arrival variable contains the bounds $\hat{\alpha}^G(\Delta)$ and $\hat{d}^G$ and the output predicate ϕ^O guarantees $\alpha(\Delta) \leq \hat{\alpha}^G(\Delta)$, and $d \geq \hat{d}^G$. The input assumption on the other hand contains the bounds $\hat{\alpha}^A(\Delta)$ and $\hat{d}^A$ and the input predicate ϕ^I reflects the assumption that $\alpha(\Delta) \leq \hat{\alpha}^A(\Delta)$ and $d \geq \hat{d}^A$.

The value of a service variable consists of a service curve $\beta(\Delta)$. The output guarantee on a service variable contains the bound $\hat{\beta}^G(\Delta)$, and the output predicate ϕ^O guarantees $\beta(\Delta) \geq \hat{\beta}^G(\Delta)$. The input assumption contains the bound $\hat{\beta}^A(\Delta)$ and the input predicate ϕ^I reflects the assumption $\beta(\Delta) \geq \hat{\beta}^A(\Delta)$ for all Δ.

In order to determine whether two Real-Time Interfaces are compatible, we need to check that $\phi^O \Rightarrow \phi^I$ is true for all connections, see section 2. Two Real-Time Interfaces that are connected only via a single arrival variable are compatible if

$$(\hat{d}^A \leq \hat{d}^G) \quad \wedge \quad (\hat{\alpha}^A(\Delta) \geq \hat{\alpha}^G(\Delta)) \quad \forall \Delta \geq 0 \tag{3}$$

and when connected only via a service connection they are compatible if

$$\hat{\beta}^A(\Delta) \leq \hat{\beta}^G(\Delta) \quad \forall \Delta \geq 0 \tag{4}$$

From this, we can generalize that two Real-Time Interfaces are compatible if (3) and (4) are true for all internal arrival and service connections respectively, and if the input predicates of all open input variables are still satisfiable.

3.5 Comment I: The Relation to Demand and Supply Bound Functions

Recently, the concept of demand bound functions **dbf**(Δ) and supply bound functions **sbf**(Δ) got about in the area of compositional scheduling, see e.g. [3], [12],[13] or [2]. With these functions, a component is considered to be schedulable, if **dbf**$(\Delta) \leq$ **sbf**$(\Delta) \forall \Delta$.

This concept also exists in the theory of Real-Time Interfaces, where the bound $\hat{\beta}^A(\Delta)$ of the input assumption on a service connection can be interpreted as a demand bound function **dbf**(Δ), and the bound $\hat{\beta}^G(\Delta)$ of the output guarantee on a service connection can be interpreted as a supply bound function **sbf**(Δ). Then, the compatibility requirement (4) on a service connection equals the above described schedulability requirement **dbf**$(\Delta) \leq$ **sbf**(Δ).

3.6 Comment II: Service Curves as Generic Resource Model

One of the major differences of this work to classical methods for scheduling analysis is the attempt to make the characteristics of resources a central part of the analysis method. This is achieved by explicit and exact modeling of available and remaining resources, and this appears to be a major prerequisite for composability and hierarchical scheduling. First, but restricted approaches towards this direction can be found in [14]. We are however going a major step forward and model not only event streams and resource availability but also propagate the associated constraints through the associated component interfaces. A similar approach was taken in [12] and [13], the work presented there is however restricted towards the use of two very basic resource models, whereas the work presented here is based on the much more generic and fine-grained resource model of service curves.

4 A Component System with Real-Time Interfaces

We distinguish three different types of system elements that are used as building blocks to build model a real-time system. Firstly, a real-time system contains a set of hardware resources, such as CPU's, DSP's or buses, that provide computing and communication services. Secondly, a real-time system contains a set of real-time load specifications, that describe how the system is being used by the environment, i.e. how much load arriving events on an incoming event-stream generate, and what the delay constraints of an event-stream are. And finally, a real-time system contains a set of processes, that use the available system services to process the incoming real-time event-streams. Different processes may share the available system resources, and the method of sharing is defined by a scheduling policy.

The following sections first describe the Real-Time Interface for service and load components and for process components with FP scheduling, as established in [15]. In sections 4.4 and 4.5 we then introduce new process components for RM and for EDF scheduling.

4.1 Service Component

A *service component* models a computing or communication resource. The Real-Time Interface of a service component has a single service output variable with the output guarantee $\phi^O = (\beta \geq \hat{\beta}^G)$.

4.2 Load Component

A *load component* models a real-time load specification. The Real-Time Interface of a load component has a single arrival output variable with the output guarantee $\phi^O = (\alpha \leq \hat{\alpha}^G) \wedge (d \geq \hat{d}^G)$.

4.3 Process Component for FP Scheduling

A *process component for preemptive fixed priority scheduling* models a process that shares system services with a fixed priority scheduling strategy. The component description of this component in Real-Time Calculus is given by the relations (1) and (2). Its Real-Time Interface has an arrival and a service input variable with the input assumption $\phi^I = (\alpha \leq \hat{\alpha}^A) \wedge (d \leq \hat{d^A}) \wedge (\beta \geq \hat{\beta}^A)$ and a service output variable with the output guarantee $\phi^O = (\beta' \geq \hat{\beta}'^G)$, and has following interface relations that were first established in [15]:

$$\hat{\beta}'^G = RT(\hat{\beta}^G, \hat{\alpha}^G) \tag{5}$$

$$\hat{\beta}^A = \max\left\{\hat{\alpha}^G(\Delta - \hat{d}^G), RT^{-\beta}(\hat{\beta}'^A, \hat{\alpha}^G)\right\} \tag{6}$$

$$\hat{\alpha}^A = \min\left\{\hat{\beta}^G(\Delta + \hat{d}^G), RT^{-\alpha}(\hat{\beta}'^A, \hat{\beta}^G)\right\} \tag{7}$$

$$\hat{d}^A = Del(\hat{\beta}^G, \hat{\alpha}^G) \tag{8}$$

4.4 Process Component for RM Scheduling

Rate monotonic scheduling is a well-known special instance of fixed priority scheduling for a set $W = \{T_i\}$ of n real-time loads that can all be described by the periodic task model $T(p_i, e_i)$, where p_i is a period and e_i is an execution

time requirement ($e_i \leq p_i$). Internally, we model a process component for RM scheduling by n load components that are connected to n process components for FP scheduling. The n load components have the output guarantee bounds $\hat{\alpha}_i^G(\Delta) = \lceil \Delta/p_i \rceil \cdot e_i$ and $\hat{d}_i^G = p_i$, and the n process components are interconnected by their service connection, ordered according to the period of their input event stream.

4.5 Process Component for EDF Scheduling

A *process component for EDF scheduling* models n processes in a real-time system that share systems services with an earliest deadline first scheduling policy. Its Real-Time Interface has n arrival and one service input variables with the input assumption $\phi^I = \bigwedge_{\forall i}((\alpha_i \leq \hat{\alpha}_i^A) \wedge (d_i \leq \hat{d^A}_i)) \wedge (\beta \geq \hat{\beta}^A)$, and a service output variable with output guarantee $\phi^O = (\beta' \geq \hat{\beta}'^G)$.

In an EDF component, the total resource demand that the n arriving event streams generate can be bounded by the sum of their arrival curves. We therefore get the resource transformation of an EDF component, if we replace α with $\sum_{\forall i} \alpha$ in (1). This already leads us to the internal interface relation to compute $\hat{\beta}'^G$ of an EDF interface.

For $\hat{\beta}^A$, we need to satisfy the delay constraints of all n arriving event streams

$$\hat{\beta}^A(\Delta) \geq \sum_{\forall i} \hat{\alpha}_i^G(\Delta - \hat{d}_i^G) \tag{9}$$

as well as the resource constraint

$$\hat{\beta}^A(\Delta) \geq \inf\left\{\beta : \hat{\beta}'^A(\Delta) = \sup_{0 \geq \lambda \geq \Delta}\{\beta(\lambda) - \sum_{\forall i} \hat{\alpha}_i^G(\lambda)\}\right\} \tag{10}$$

The tightest $\hat{\beta}^A$ is the maximum of both expressions.

For $\hat{\alpha}_j^A$, we first need to know, which share of the resource $\hat{\beta}^G$ is available to process the arriving event stream j in an EDF component. From (9) and (3) and (4) we know that

$$\sum_{\forall i} \hat{\alpha}_i^A(\Delta - \hat{d}_i^G) \leq \hat{\beta}^G(\Delta) \tag{11}$$

and if we look at $\hat{\alpha}_j^A$ of a single event stream j, we get

$$\begin{aligned} \hat{\alpha}_j^A(\Delta - \hat{d}_j^G) &\leq \inf_{0 \leq \lambda}\{\hat{\beta}^G(\Delta + \lambda) - \sum_{\forall i \neq j} \hat{\alpha}_i^A(\Delta - \hat{d}_i^G + \lambda)\} \\ &\stackrel{def}{=} \hat{\beta}_{EDF,j}^G \end{aligned} \tag{12}$$

The right side of (12) equals the share of the resource $\hat{\beta}^G$ that is available to process the arriving event stream j.

For $\hat{\alpha}_j^A$ we then need to satisfy the delay constraint

$$\hat{\alpha}_j^A(\Delta) \leq \hat{\beta}_{EDF,j}^G(\Delta + \hat{d}^G) \tag{13}$$

and the resource constraint

$$\hat{\alpha}_j^A(\Delta) \leq \sup\left\{\alpha : \hat{\beta}'^A(\Delta) = \sup_{0 \geq \lambda \geq \Delta}\{\hat{\beta}_{EDF,j}^G(\lambda) - \alpha(\lambda)\}\right\} \tag{14}$$

The tightest $\hat{\alpha}_j^A$ is the minimum of both expressions.

We can then establish the set of interface relations in a process component for EDF scheduling:

$$\hat{\beta}'^G = RT(\hat{\beta}^G, \sum_{\forall i} \hat{\alpha}_i^G) \tag{15}$$

$$\hat{\beta}^A = \max\left\{\sum_{\forall i} \hat{\alpha}_i^G(\Delta - \hat{d}_i^G), RT^{-\beta}(\hat{\beta}'^A, \sum_{\forall i} \hat{\alpha}_i^G)\right\} \tag{16}$$

$$\hat{\alpha}_j^A = \min\left\{\hat{\beta}_{EDF,j}^G(\Delta + \hat{d}_j^G), RT^{-\alpha}(\hat{\beta}'^A, \hat{\beta}_{EDF,j}^G)\right\} \tag{17}$$

$$\hat{d}_j^A = Del(\hat{\beta}_{EDF,j}^G, \hat{\alpha}_j^G) \tag{18}$$

with

$$\begin{aligned} RT^{-\alpha}(\beta', \beta)(\Delta) &= \beta(\Delta + \lambda) - \beta'(\Delta + \lambda) \\ &\text{for} \quad \lambda = \sup\{\tau : \beta'(\Delta + \tau) = \beta'(\Delta)\} \end{aligned} \tag{19}$$

and

$$\begin{aligned} RT^{-\beta}(\beta', \alpha)(\Delta) &= \beta'(\Delta - \lambda) + \alpha(\Delta - \lambda) \\ &\text{for} \quad \lambda = \sup\{\tau : \beta'(\Delta - \tau) = \beta'(\Delta)\} \end{aligned} \tag{20}$$

5 Real-Time Interfaces for Servers

Server based scheduling is widely used in the area of real-time system design to enable hierarchical scheduling of different system components. With the component system introduced in the previous section, we may already model systems where various subsystems with internal FP, RM or EDF scheduling hierarchically share one resource using a top-level fixed priority scheduling policy. To further extend the component system's ability of modeling hierarchical scheduling, the following sections introduce Real-Time Interfaces of components that model polling servers (PS). Polling servers are only used exemplary here. Real-Time Interfaces for other server types could be derived very similarly.

A polling server can be thought of as a periodic task $T(p, e)$. When the task T is selected to run by the scheduler, it checks whether workload is waiting to be processed by the server. If yes, the server will provide e resources to process the waiting workload. But if no work is available for the server, the task will immediately be finished, i.e. the server will not check for arriving work anymore until the next period starts.

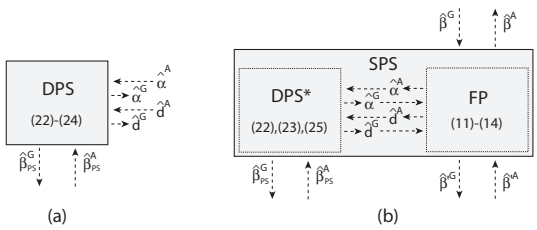

Figure 6. Real-Time Interfaces of a dynamic (a), and a static (b) PS component.

5.1 Polling Server for Dynamic Scheduling

A polling server for dynamic scheduling (DPS) can be implemented by a periodic task $T(p, e)$ with an associated deadline $d = p$. For an EDF component, a polling server behaves like an arriving event stream with arrival curve $\alpha(\Delta) = \lceil \Delta/p \rceil \cdot e$ and a maximum delay $d = p$, and to components connected to the service output, the server is a resource that provides a resource supply of e during every interval of length p. We have however to consider two specialities. Firstly, a server might not provide any resources during one period. This happens if workload arrives just after the server task was selected to run by the scheduler and was finished because there was no workload waiting to be processed. And secondly, under EDF scheduling we do not know when the server task receives the e resources within a period, and the DPS can therefore only guarantee to supply e resources at the end of every period p. We can establish the set of interface relations in a DPS as:

$$\hat{\alpha}^G = \left\lceil \frac{\Delta}{p} \right\rceil e \tag{21}$$

$$\hat{d}^G = p \tag{22}$$

$$\hat{\beta}^G_{DPS} = \max\left\{0, \left\lfloor \frac{\Delta - p}{p} \right\rfloor e\right\} \tag{23}$$

5.2 Polling Server for Static Scheduling

A polling server for static scheduling (SPS) can be implemented by a periodic task $T(p, e)$ with an associated maximum allowable delay $d = p$. Internally, the Real-Time Interface of a polling server for static scheduling can be modeled by a slightly modified version of a polling server for dynamic scheduling and a process component for FP scheduling. The difference to a polling server for dynamic scheduling is that we know when the server task receives the e resources at the latest within a period. A SPS can therefore guarantee a slightly better service. Considering this, the interface relation for $\hat{\beta}^G_{PS}$ can be established as:

$$\hat{\beta}^G_{SPS} = \max\left\{0, \left\lfloor \frac{\Delta - p}{p} \right\rfloor e + \min\left\{e, \hat{\beta}^G\left(\Delta - \left\lfloor \frac{\Delta}{p} \right\rfloor p\right)\right\}\right\} \tag{24}$$

Besides this, the interface relations for $\hat{\beta}'^G$, $\hat{\beta}^A$, $\hat{\alpha}^A$, $\hat{d}^A$, $\hat{\alpha}^G$ and $\hat{d}^G$ remain unchanged and are given by (5), (6), (7), (8), (21) and (22), respectively.

5.3 Feeding Back Unused Resources

Relation (21) validly upper bounds the workload $\hat{\alpha}^G$ that polling servers generate on a system. But this bound is typically overly pessimistic, because polling servers will not claim resources if no workload is present to be processed by them. Hence, the workload $\alpha(\Delta)$ that a PS generates on a system is not only upper bounded by (21), but also by the sum the of the workloads $\alpha_i(\Delta + p)$ of all process components that are processed by the server and that are not hierarchically decoupled by another server:

$$\alpha(\Delta) \leq \min\left\{\left\lceil \frac{\Delta}{p} \right\rceil e, \sum_{i \in PS} \alpha_i(\Delta + p)\right\} = \min\left\{\left\lceil \frac{\Delta}{p} \right\rceil e, \Sigma(\Delta + p)\right\} \tag{25}$$

To consider this new bound in the RTI of polling servers, we extend the value of a service variable in Real-Time Interfaces by a new curve $\Sigma(\Delta)$, that represents the sum of all workloads $\alpha_i(\Delta)$ that are processed with the service $\beta(\Delta)$ of the service variable and that are not hierarchically decoupled by a polling server. The output guarantee and input assumption on a service variable contain then additionally the bounds $\Sigma^G(\Delta)$ and $\Sigma^A(\Delta)$, respectively, with the relation $\hat{\Sigma}^A(\Delta) \leq \hat{\Sigma}^G(\Delta) \leq \Sigma(\Delta)$ that must be true for a compatible service connection of two interfaces.

To obtain the internal interface relation for this new bound Σ^G, we need to satisfy the resource constraint

$$\Sigma^G \leq \sup\left\{\Sigma \;:\; \hat{\alpha}^G(\Delta) = \min\left\{\left\lceil \frac{\Delta}{p} \right\rceil e, \Sigma(\Delta + p)\right\}\right\} \tag{26}$$

We can therefore establish the interface relations

$$\hat{\alpha}^G = \min\left\{\left\lceil \frac{\Delta}{p} \right\rceil e, \hat{\Sigma}^A\right\} \tag{27}$$

$$\hat{\Sigma}^G(\Delta + p) = \min^{-1}\left\{\hat{\alpha}^A, \left\lceil \frac{\Delta - p}{p} \right\rceil e\right\} \tag{28}$$

with

$$\min^{-1}(\alpha_1, \alpha_2) = \alpha_1(\Delta + \lambda) \quad \text{for} \quad \lambda = \inf\{\tau \;:\; \alpha_1(\Delta + \tau) \leq \alpha_2(\Delta + \tau)\} \tag{29}$$

While (27) replaces (21) in the already established interface relations for polling servers, (28) complements the established interface relations.

Note, that since we extended the value of a service variable in Real-Time Interfaces by the new curve $\Sigma(\Delta)$, we need to establish interface relations for $\hat{\Sigma}'^G$ and $\hat{\Sigma}^A$ for all process components of our component system. Further, we must also adapt the interface relations for $\hat{\alpha}^A$ in the process component interfaces. For the FP process component interface, we get $\hat{\Sigma}'^G = \hat{\Sigma}^G - \hat{\alpha}^G$ and $\hat{\Sigma}^A = \hat{\Sigma}'^A + \hat{\alpha}^G$. And to get $\hat{\alpha}^A$ we must now compute the minimum of (7) and the additional new bound $(\hat{\Sigma}^G - \hat{\Sigma}'^A)$. For the other process components, the extensions can be derived similarly.

6 Application

In this section we will look at some applications that Interface-based Design with Real-Time Interfaces enable. For this, we will use an imaginary real-time system with a complex mix of hierarchically arranged static and dynamic scheduling. The level of complexity of the chosen system may seem unrealistic, but was chosen on purpose, to include many different scheduling varieties in one system.

6.1 Case Study System Specification

The case study real-time system consists of a set of 10 real-time tasks *T1-T10*, that are running on a processor under a scheduling hierarchy according to Fig. 7. The scheduling hierarchy uses a mix of FP, RM and EDF scheduling

as well as a polling server for static scheduling ($p_{SPS} = 4, e_{SPS} = 1$) and a polling server for dynamic scheduling ($p_{DPS} = 3, e_{DPS} = 1$). The processor is fully available to process the 10 tasks, and all tasks are fully preemptable and independent from each other. The tasks are specified in Table 1 with a period *P*, a maximum jitter *J*, a minimum inter-arrival distance *D* (only applicable if $J + D > P$), a maximum allowable delay *d* (relative deadline), and a worst-case execution time *e*. Further, the right most column of Table 1 lists the average system utilization of the different tasks. Note, that the task set contains some periodic tasks (*T1*, *T3*, *T6-T10*), but also tasks with jitter (*T2*, *T4*), as well as a task with burst (*T5*). Further, some tasks have relative deadlines that are shorter than their period (*T1*, *T3*), while others have relative deadlines that are longer than their period (*T4*, *T5*, *T6*).

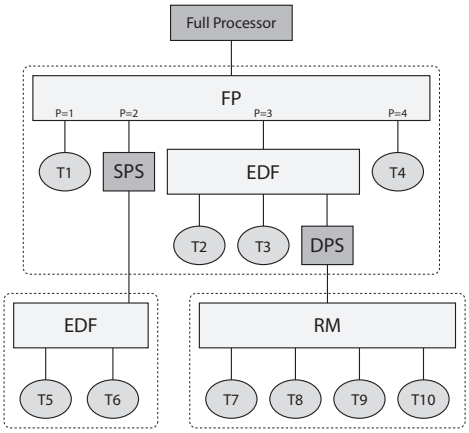

Figure 7. Hierarchical scheduling scheme.

Table 1. Real-Time Load Specifications

Load	$\hat{\alpha}^G(P, J, D)$	$\hat{d}^G$	e	øU
T1	(5, 0, 0)	2	0.2	0.04
T2	(10, 5, 0)	10	1	0.1
T3	(25, 0, 0)	15	1.5	0.06
T4	(100, 2, 0)	150	40	0.4
T5	(25, 80, 5)	50	2	0.08
T6	(20, 0, 0)	30	2	0.1
T7	(12, 0, 0)	12	0.5	0.042
T8	(16, 0, 0)	16	0.75	0.047
T9	(20, 0, 0)	20	1	0.05
T10	(30, 0, 0)	30	2	0.067
Total				0.985

6.2 Real-Time Interface System Model

For interface-based design and analysis, we first need to specify the Real-Time Interfaces of all system components. To model the processor, we use a service component interface with a bound $\hat{\beta}^G(\Delta) = \Delta$. We further model the tasks using load components with the bounds $\hat{d}^G = d$ and $\hat{\alpha}^G$. We can thereby use (30) to obtain $\hat{\alpha}^G$ from the (P, J, D) specification.

$$\{(P, J, D), e\} \;\Rightarrow\; \hat{\alpha}^G = \min\left\{\left\lceil \frac{\Delta + J}{P} \right\rceil e, \left\lceil \frac{\Delta}{D} \right\rceil e\right\} \qquad (30)$$

With these service and load components and with the appropriate process and polling server components, we then compose the system interface model as depicted in Fig. 8.

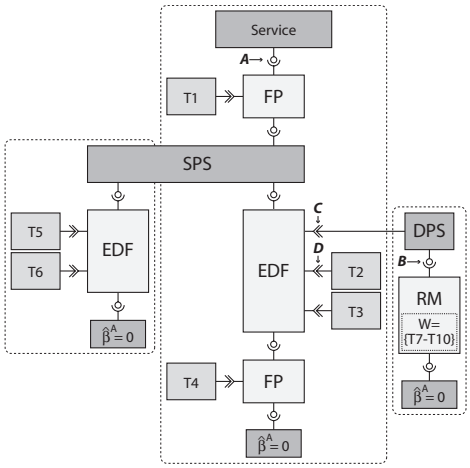

Figure 8. RTI model of the case study system.

6.3 Interface-Based Schedulability Analysis

With Real-Time Interfaces, schedulability analysis is done implicitly during system composition. And since we can successfully compose all component interfaces of the case study (depicted in Fig. 8), we are guaranteed that the system as specified is schedulable.

Let us now look at the service connection that connects the processor service component interface to the interface of the rest of the system. This connection is labeled ***A*** in Fig. 8, and the A/G bounds at this connection are depicted in Fig. 9. As expected, we see that the service assumption $\hat{\beta}^A_{tot}$ at this connection is smaller than the service guarantee $\hat{\beta}^B_{tot}$, which complies to the requirement (4) for interface compatibility.

From the definition of Real-Time Interfaces, we know that $\hat{\beta}^A_{tot}$ can be interpreted as the demand bound function **dbf** of the case study system, since it specifies the minimum assumed service that is required in any time interval to make the complete system schedulable. And according to (4), we can guarantee that the system is then schedulable on any resource with a service guarantee (or supply bound function **sbf**) $\hat{\beta}^G_{tot} \geq \hat{\beta}^A_{tot}$. This means that once we know $\hat{\beta}^A_{tot}$, we can directly decide whether our system is schedulable on a resource with a given service guarantee $\hat{\beta}^G_{tot}$. In particular we do not need to do any schedulability analysis, other than checking that $\hat{\beta}^G_{tot} \geq \hat{\beta}^A_{tot}$, i.e. the complete system schedulability information is encoded in $\hat{\beta}^A_{tot}$.

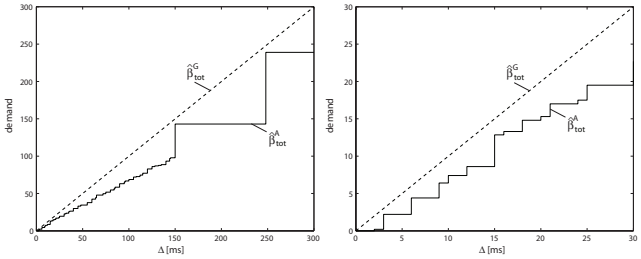

Figure 9. A/G bounds at connection *A*.

6.4 Interface-Based Design

Interface-based design of real-time systems benefits from all advantages of interface-based design as described in [9]. Most notable, it supports *incremental design* and *independent implementability* of real-time systems.

The property of incremental design allows to design an economic real-time system by first composing all load and process components. This leads to an interface model with only one open service input. By looking at the service input assumption $\hat{\beta}^A$ on this open input, we can directly find the tightest possible service interface with $\hat{\beta}^G \geq \hat{\beta}^A$. Then, by choosing the most economic resource, e.g. processor, that still conforms to this tight service guarantee, we obtain an economic real-time system that guarantees system schedulability, without being over-dimensioned.

This design procedure stands in contrast to traditional system design, where resource components are chosen a-priori, and performance analysis methods like [11] or [14] are used a-posteriori to decide whether a system is schedulable or not. In this traditional approach, economic designs must be found by trial-and-error, i.e. by parameter sweeps or binary search.

6.5 Unused Resources in Polling Servers

Let us next take a closer look at the polling server for dynamic scheduling *DPS* in the case study system. On the left side of Fig. 10, the A/G bounds at the service connection (***B***) between the DPS and the RM component are depicted.

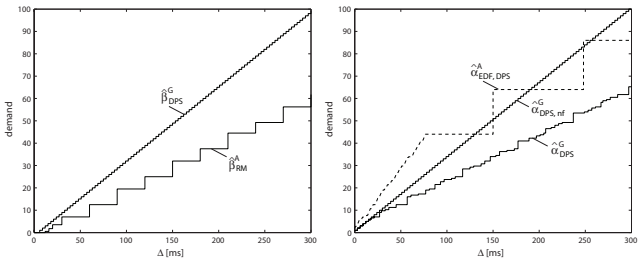

Figure 10. A/G bounds at *B* (left), and *C* (right).

On the right side of Fig. 10 on the other hand, the A/G bounds at the arrival connection (***C***) between the DPS and the EDF component are depicted. Here, $\hat{\alpha}^G_{DPS,nf}$ is the guarantee bound of the polling server, if feeding back of unused resources is not factored in. To be schedulable, we see that the maximum load that the polling server generates on the EDF process component must be smaller or equal $\hat{\alpha}^A_{EDF,DPS}$.

In recent work on hierarchical scheduling, hierarchy is typically achieved by servers that are implemented as simple periodic tasks with period p_{PS} and execution time e_{PS}, see e.g. [12] or [13]. Such an implementation would generate a load with an arrival guarantee as defined by (21). In Fig. 10, $\hat{\alpha}^G_{DPS,nf}$ depicts this arrival guarantee, and as we see, this guarantee does not comply with the arrival assumption $\hat{\alpha}^A_{EDF,DPS}$ of the EDF process component. Therefore, such a polling server implementation would render our system to be not schedulable.

In reality however, polling servers will not claim resources if no workload is present to be processed by them. This is factored in by the improved arrival guarantee for polling servers, as defined in (25). This improved arrival guarantee $\hat{\alpha}^G_{DPS}$ is also depicted Fig. 10, and as we see, this guarantee complies with the arrival assumption $\hat{\alpha}^A_{EDF,DPS}$ of the EDF process component.

6.6 Interface-Based System Adaption

After composition of the complete system, the assume bounds on all internal component connections specify the maximum arrival load and minimum service, respectively, that is allowable at the specific connection to keep the system schedulable. We can exploit this information to adapt the load on our system up to the maximum limit without trial-and-error, and therefore without the danger of rendering the system to become unschedulable. Similarly, we could also directly reduce the service to our system down to the minimum, as we already explained in section 6.4.

In the left side of Fig. 11, $\hat{\alpha}^A_{EDF,T2}$ depicts the arrival assumption of the EDF process component to the load arrival of task *T2* at connection ***D***. We see, that the current arrival guarantee $\hat{\alpha}^G_{T2}$ of task *T2* does by far not exploit this assumption. We can therefore for example increase the burstiness of *T2* by changing its timing specification from $(10, 5, 0)$ to $(10, 30, 1)$. Since the new arrival guarantee $\hat{\alpha}^{G*}_{T2}$ still complies with $\hat{\alpha}^A_{EDF,T2}$, we are guaranteed that the system stays schedulable after this load adaption. The right side of Fig. 11 depicts the influence of this change to the service assumption of the complete system at connection ***A***.

Alternatively, when we compute the delay assumption $\hat{d}^A_{EDF,T2}$ of the EDF process component to the task *T2*, we get $\hat{d}^A_{EDF,T2} = 3.2$. Therefore, we know that we could reduce the relative deadline of *T2* from currently 10 down to 3.2. Note, that $\hat{\alpha}^A_{EDF,T2}$ will change when we reduce the

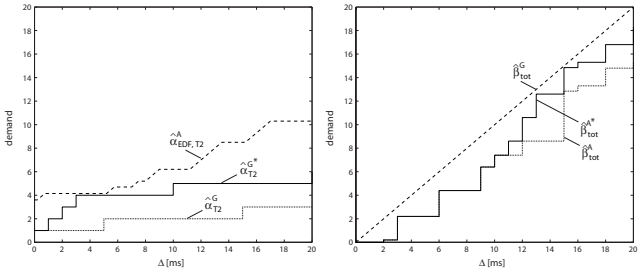

Figure 11. A/G bounds at *D* (left) and *A* (right).

delay guarantee $\hat{d}^{G}_{T2}$. Figure 12 depicts the changes on the bounds at the connections ***D*** (left) and ***A*** (right).

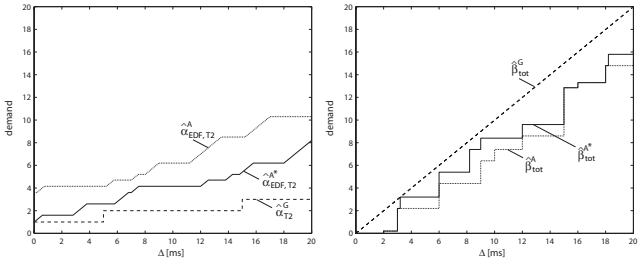

Figure 12. A/G bounds at *D* (left) and *A* (right).

6.7 Computational Complexity

To analyze the case study system, we used a prototype implementation of Real-Time Calculus and Real-Time Interfaces. This prototype is implemented in Java and uses Matlab as user frontend. The implementation internally uses an efficient representation for Variability Characterization Curves that allows to represent VCC's with $\Delta \in \mathbb{R}^{\geq 0}$ without the need of linear approximation. All required curve operations are implemented to work on these efficiently represented VCC's.

With this prototype tool, it took less than $1s$ to compose (and analyze) the complete case study system using Matlab 7 on a Pentium Mobile 1.6 GHz.

For applications, where computation needs to be faster (e.g. exploration) or where not so much computational power is available (e.g. online service/load adaption or admittance tests), linear approximated VCC's could be used, that trade off computational complexity with the tightness of the obtained bounds, see e.g. [1].

7 Conclusions

In this work, we extended the theory of Real-Time Interfaces and we introduced a component system with Real-Time Interfaces that defines the building blocks for interface-based design of real-time systems with hierarchically mixed static and dynamic scheduling. For this, we defined the Real-Time Interfaces of components that model hardware resources, real-time loads (event-streams) and software-processes for preemptive FP scheduling, RM scheduling as well as EDF scheduling. Further, we defined the Real-Time Interfaces of components that model polling servers for static and dynamic scheduling. These interfaces factor in that polling servers do not claim resources if no workload has to be processed. We have shown the applicability of interface-based design and analysis with Real-Time Interfaces in a case study of a real-time system with complex hierarchical static and dynamic scheduling.

References

[1] K. Albers and F. Slomka. An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*. IEEE Press, 2004.

[2] L. Almeida and P. Pedreira. Scheduling within temporal partitions: response-time analsysis and server design. In *Proceedings of the Fourth ACM International Converence on Embedded Software*, 2004.

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

[4] E. Bini and G. Lipari. Resource partitioning among real-time applications. In *Proc. of EUROMICRO Conference on Real-Time Systems (ECRTS)*, 2003.

[5] A. Chakrabarti, L. de Alfaro, T. Henzinger, and M. Stoelinga. Resource interfaces. In *EMSOFT 03: Embedded Software*, Lecture Notes in Computer Science 2855, pages 117–133. Springer-Verlag, 2003.

[6] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design, Automation and Test in Europe (DATE)*, pages 190–195, March 2003.

[7] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 148–165. Springer-Verlag, 2001.

[8] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[9] L. de Alfaro and T. A. Henzinger. Interface-based design. In *To appear in the Proceedings of the 2004 Marktoberdorf Summer School*. Kluwer, 2005.

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

[11] K. Richter, M. Jersak, and R. Ernst. A formal approach to mpsoc performance verification. *IEEE Computer*, 36(4):60–67, April 2003.

[12] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 2–13. IEEE Press, 2003.

[13] I. Shin and I. Lee. Compositional Real-Time Scheduling Framework. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 57–67. IEEE Press, 2004.

[14] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[15] E. Wandeler and L. Thiele. Real-Time Interfaces for Interface-Based Design of Real-Time Systems with Fixed Priority Scheduling. In *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05)*, pages 80–89. IEEE Press, 2005.

[16] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-time component-based systems. In *Proceedings of the 11th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 428–437. IEEE Press, 2005.