An Interface Algebra for Real-Time Components *

Thomas A. Henzinger
EPFL and UC Berkeley
tah@epfl.ch

Slobodan Matic
UC Berkeley
matic@eecs.berkeley.edu

Abstract

We present an assume-guarantee interface algebra for real-time components. In our formalism a component implements a set of task sequences that share a resource. A component interface consists of an arrival rate function and a latency for each task sequence, and a capacity function for the shared resource. The interface specifies that the component guarantees certain task latencies depending on assumptions about task arrival rates and allocated resource capacities. Our algebra defines compatibility and refinement relations on interfaces. Interface compatibility can be checked on partial designs, even when some component interfaces are yet unknown. In this case interface composition computes as new assumptions the weakest constraints on the unknown components that are necessary to satisfy the specified guarantees. Interface refinement is defined in a way that ensures that compatible interfaces can be refined and implemented independently. Our algebra thus formalizes an interface-based design methodology that supports both the incremental addition of new components and the independent stepwise refinement of existing components. We demonstrate the flexibility and efficiency of the framework through simulation experiments.

1 Introduction

The increasing complexity of real-time and embedded systems necessitates advanced design and maintenance procedures for the assurance of timing requirements. Automatic tools are highly desired for such an error-prone and tedious process. Since timing performance has to be estimated for a large number of design alternatives, the tools are required to be efficient. In addition to that, common modification of system requirements demands flexible procedures.

An often advocated software engineering approach is *component-based* design, in which a software system is created by combining preexisting modules with new software that provides both glue between the components, and new functionality. This simplifies the design process since such a system decomposition provides a solution to the original large problem by solving several smaller problems. Another advantage of such an approach lies in the fact that component performance analysis can detect design errors before the components are implemented and composed.

Previous research in component-based real-time systems concentrated on partitioning and scheduling frameworks that make the temporal behavior of a component independent of the presence of other components in the system [5, 11]. More recent works present methods that abstract the internal complexity of a real-time component into a component *interface* that is subsequently used for the rest of the design [12, 7, 1]. This research considers the *periodic resource* model (T, C), a resource abstraction under which a component is guaranteed to get C units of the resource every T units of time. The methods show how to abstract a set of independent periodic tasks with EDF or RM scheduling algorithms into a single periodic task. Later work [13] shows how to abstract a set of independent periodic tasks into a *bounded-delay* interface. The bounded-delay resource model (c, δ), studied in [10, 8, 9], guarantees fraction c of the resource with at most δ time units of delay.

In this paper we start with a different task group model and use a method, similar to the one presented in [13], for abstracting such a group into a bounded-delay interface. The task model consists of a set of aperiodic tasks each specified with an arrival rate function and a relative deadline. The arrival rate function bounds the number of task requests in a given interval of time. To abstract such a task group we consider only the bounded-delay resource model with EDF scheduling, although the results for the periodic resource model can be applied in a similar setting. Then we consider such a task group as a part, i.e., a component, of a larger real-time system specified with a set of task sequences that define task precedence constraints. The objective of the paper is to enable the automatic, efficient, and flexible component-based design of such a system.

To address the problem we apply concepts from interface

*This research was supported by the AFOSR MURI grant F49620-00-1-0327 and by the NSF grants CCR-0225610 and CCR-0208875.

theories [2, 3]. In this formalism an interface of a component specifies what the component expects (assumes) from its environment and what it provides back (guarantees) to the environment. The constraints should be sufficient to check if two interfaces are *compatible*, i.e., if the underlying components work properly when composed together. In the real-time context 'proper' means satisfying timing requirements, e.g., end-to-end latency. Since the system specification includes dependencies between the tasks from different components, the interface cannot just contain resource constraints as in previous works, but also dataflow propagation constraints. Therefore, besides the resource requirements, an interface also specifies the arrival rate assumption and the latency guarantee for task sequences.

We define an *interface algebra* for real-time interfaces, a formal algebra that enables tool support for the proposed methodology. Besides the compatibility relation, the algebra consists of two operations and a relation. The interface *composition* operation collects two interfaces and adds the resource requirements of the underlying components. The interface *connection* operation enables sequencing of interface tasks. The refinement relation aims at formalizing the relation between abstract and concrete versions of the same component. A more refined version of a component makes weaker input assumptions and stronger output guarantees than a more abstract description. Therefore, in a design we can always substitute a refined version for an abstract one.

One of the beneficial properties of the interface formalism is *incremental design*. According to this property the composition of interfaces can be performed in any order, i.e., it is associative. Besides having a more flexible framework, this also means being able to check compatibility and compute the composition of the two interfaces without specifying the interfaces of other components in the system. Note that task group abstraction procedures are generally not associative.

Additionally, in component-based design, one wants to refine an interface towards an implementation, independently of the implementation of other components. If all implementations satisfy their respective interfaces, the components should properly work together. The *independent implementability* property of the formalism states that in order to refine a given composition of two interfaces, it suffices to independently refine each interface and then compose the obtained refinements. This property enables the system correctness to be established during interface design, without global checks after the components are implemented.

Our formalism supports automatic interface compatibility and interface refinement checking. We introduce interfaces as stateless objects that are represented by predicates. Thus, checking of the two properties is efficient. We also consider more expressive interfaces that describe components with performance polymorphism, i.e., with multiple levels of service. In this paper we are concerned with defining the algebra and showing how it can be used on a few examples of real-time applications of moderate complexity.

Related work. Besides the theoretical work in compositional real-time scheduling frameworks, the increased interest in real-time component-based systems has recently resulted in first implementations. In [15] the interface of a software component is extended to include real-time assumptions and guarantees of the component. We use similar functional and temporal specifications, except that we allow for multiple levels of service of a component. Since the goal of [15] is reusability across different platforms, the resource consumption specification is not part of the component interface. So, the resource utilization computation is separated from the application design, which assumes virtual resources. Besides, no abstraction of resource requirements is studied.

The approach taken in this paper is most similar to the recent work [14]. That work is the first research effort that formally combines the network calculus and interface design theories in the real-time context. It is not limited to a particular task set characterization or to a particular resource model. In contrast to the traditional real-time approaches, it allows for the composition of software process components before the hardware resource components are specified. In [14] each component represents a task. There is no abstraction of task groups into components, and no discussion of interface refinement, which is one of the goals of our work. Also, the task model in [14] assumes independent tasks, so interface compatibility checking does not have to take into account dataflow constraints. Finally, they assume preemptive fixed-priority scheduling, where each component is specified with a certain priority.

In interface theory research [2, 4] the component interaction is specified using expressive interfaces. The temporal input/output behavior of a component is typically captured by an automaton. Therefore, the automaton of the composite interface is constructed by pruning all violating states from the product of the component automata. Such a stateful approach is a more general way to address multiple levels of component performance. However, in this paper we keep the interface formalism simple in order to focus more on real-time issues.

Outline. Sec. 2.1 introduces the real-time component model studied in the paper, with its functional, temporal, and resource parts. The temporal portion of the component interface consists of an arrival rate function and a delay for each request. The same section introduces the resource portion of the interface in the form of the bounded-delay resource model. How to obtain resource partition parameters for a group of tasks is presented in Sec. 2.2. We introduce interfaces in Sec. 3.1, and formally define an interface algebra in Sec. 3.2. A discussion of how interfaces can be

used for efficient and automatic component-based design and verification is given in Sec. 4. In particular, incremental design is discussed in Sec. 4.1 and independent implementability in Sec. 4.2.

2 Real-Time Components

2.1 Task and Resource Model

Functional model. Let a *task sequence* $\pi = \tau_1\tau_2\ldots\tau_k$ be a finite sequence of tasks. There exists a precedence constraint between τ_j and τ_{j+1} for each $j = 1,\ldots,k-1$. Although our arguments can be generalized for trees of tasks, we keep the task sequence model for simplicity reasons. We consider components as units for the implementation, reuse, and composition of task sequences. The functional description of a component consists of a set of task sequences. Two task sequences from the same component can contain the same task. Fig. 1 shows an example of a component with two task sequences, $\tau_1\tau_3$ and $\tau_2\tau_3$. The task sequences are interleaved and independent, and tasks are preemptible. For the purposes of the paper it is not important whether task inputs/outputs are data processed by tasks or only requests for task execution.

Arrival-delay temporal model. The temporal interface of a component is similar to the interface of a component in [15], and consists of an arrival rate function and a maximum delay for each sequence of the component. In general, it consists of several pairs of arrival rate functions and delays, one pair for each level of service of the component, as further discussed in Sec. 4.

An *arrival rate function* a for a task sequence is a function that bounds the number of the invocations of the task sequence: for a time interval of length t the number of invocations is bounded by $a(t)$. In this paper we concentrate on the *bursty arrival* pattern, which is defined by the function $a(t) = \sigma + \rho\cdot t$ for some $\sigma, \rho \in \mathbb{R}_{\geq 0}$. For instance, sporadic invocation patterns can be modeled with bursty arrival rate functions. The expression for a gives the upper bound on the number of invocations. When required we consider the integer upper bound $\lfloor a(t)\rfloor$.

A number $d \in \mathbb{R}$ is a *delay* for a task sequence if all tasks of the sequence must be completed within d units of time, i.e., a sequence output must be generated at most d time units after the occurrence of a sequence input.

Bounded-delay resource model. Let the *capacity* $0 \leq c \leq 1$ be a fraction of the resource assigned to a component, and let $\delta \geq 0$ be the maximum time the component may have to wait to receive this fraction. A resource is called a *bounded-delay resource* $R = (c,\delta)$ if for every real $L > 0$, it can guarantee allocations of at least $c \cdot L$ units of the resource in every time interval of length $L + \delta$ [10].

The motivation for the bounded-delay resource model comes from the fact that the resource demand of a component cannot be precisely described only with a resource fraction c. This is so, because different components may have considerably different delay requirements [9]. The choice of the delay bound δ addresses the trade-off between high context switch costs (smaller δ) and high task execution latencies (larger δ).

For a given bounded-delay resource $R = (c,\delta)$, the resource *supply bound function* $\mathtt{sbf}_R$: $\mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ maps each $t \in \mathbb{R}_{\geq 0}$ to the minimum supply of the resource R over all time intervals of length t. From the definition of the bounded-delay resource model it follows that

$$\mathtt{sbf}_R(t) = \mathtt{sbf}_{(c,\delta)}(t) = \begin{cases} 0 & \text{if } t \leq \delta, \\ c(t-\delta) & \text{if } t > \delta. \end{cases}$$

2.2 Task and Component Composition

Task composition. We first briefly review the results from [12, 13] for schedulability conditions under the bounded-delay model and EDF scheduling. Then we apply and generalize them for the task model used in this paper.

Let W be a set of independent and preemptive tasks that share the same resource, and let R be a bounded-delay resource model. We say that (W, R, EDF) is *schedulable* if under every instance of allocations of the resource R, there exists a feasible EDF schedule for W [12]. If (W, R, EDF) is schedulable, then the set of tasks W under the resource $R = (c,\delta)$ and the EDF scheduling algorithm can be abstracted as a single requirement (c,δ), i.e., no global knowledge of task internals is necessary. A discussion on how to schedule several (c,δ) resource requirements can be found in [9].

Let W be a set of periodic tasks $\tau_j = (p_j, e_j)$, where p_j is the period, e_j is the worst-case execution time (wcet) requirement of the task τ_j, and the deadline of each task is assumed to be equal to its period. For a given set of tasks W, the resource *demand bound function* $\mathtt{dbf}$: $\mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ maps each $t \in \mathbb{R}_{\geq 0}$ to the maximum resource demand over all time intervals of length t. For the EDF scheduling algorithm we have $\mathtt{dbf}_W(t) = \sum_{\tau_j \in W} \lfloor t/p_j \rfloor \cdot e_j$.

For the case of periodic workloads W, Thm. 1 in [13] gives a sufficient and necessary condition for schedulability: (W, R, EDF) is schedulable iff for all $0 < t \leq 2\cdot \mathtt{lcm}_W$, the maximal resource demand is no greater than the minimum resource supply, i.e., $\mathtt{dbf}_W(t) \leq \mathtt{sbf}_R(t)$. In this condition, $\mathtt{lcm}_W$ is the least common multiple of the periods in W.

Finally, Thm. 3 in [13] gives a general schedulability condition for the case of other workload models W for which $\mathtt{dbf}_W$ can be computed: (W, R, EDF) is schedulable iff for all $t > 0$, we have $\mathtt{dbf}_W(t) \leq \mathtt{sbf}_R(t)$.

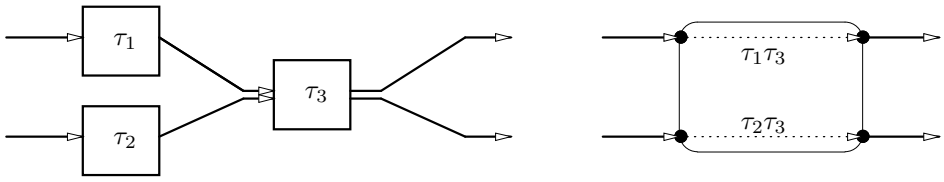

Figure 1. Task sequence and component representation

W	τ_1	τ_2	τ_3
σ	1	1	3
ρ	1/2	1/3	1/2+1/3
d	2/3	2	1
e	0.1	0.3	0.1

Table 1. Temporal interface for Fig. 1

We apply this result for the case of the aperiodic workload defined with arrival rate functions and delays for tasks. Let W be a set of tasks $\tau_j = (a_j, d_j, e_j)$, where a_j is the arrival rate function, d_j the delay, and e_j the wcet of the task τ_j, and let $R = (c, \delta)$ be the bounded-delay resource model. To apply the theorem we first compute the demand bound function of the task τ_j. We note that there are at most $\lfloor a_j(t - d_j) \rfloor$ invocations of the task τ_j that are released and required to complete in a time interval of length t. Therefore, we have

$$\mathtt{dbf}_{\tau_j}(t) = \begin{cases} 0 & \text{if } t \leq d_j, \\ \lfloor a_j(t - d_j) \rfloor \cdot e_j & \text{if } t > d_j. \end{cases}$$

The demand bound function of the total workload set W is $\mathtt{dbf}_W(t) = \sum_{\tau_j \in W} \mathtt{dbf}_{\tau_j}(t)$. Thus, both $\mathtt{dbf}_W$ and $\mathtt{sbf}_R$ are known, and we can apply Thm. 3 [13] to check if (W, R, EDF) is schedulable.

Given the task set W, let c_W be the *capacity function* that maps each bounded delay $\delta \geq 0$ to the smallest resource fraction $c_W(\delta)$ such that the component (W, R, EDF) is schedulable with $R = (c_W(\delta), \delta)$. Tab. 1 shows an instance of the task workload of the component in Fig. 1, with each task modeled as a bursty arrival task. Fig. 2 shows capacity functions c_W for each $W = \{\tau_j\}$ consisting of only a single task $\tau_j = (a_j, d_j, e_j)$. For such a simple set W, the analytical expression for c_W can be derived. For instance, we have $c_W(0) = \max\{e_j \cdot \rho_j \ , \ \sigma_j \cdot e_j/d_j\}$, and $\delta_1 = \mathtt{inv}(c_W)(1) = d_j - \sigma_j \cdot e_j$.

In the rest of the paper we assume the following notation. For two functions g_1 and g_2, with a domain set X and range set $\mathbb{R}$, and for $\phi \in \{<, \leq, >, \geq\}$, we write $g_1 \ \phi \ g_2$, if $g_1(x) \ \phi \ g_2(x)$ for all $x \in X$. For instance, $g_1 > 1$ means $g_1(x) > 1$ for all $x \in X$. Similarly, the function $g_1 + g_2$ is defined by $(g_1 + g_2)(x) = g_1(x) + g_2(x)$ for all $x \in X$.

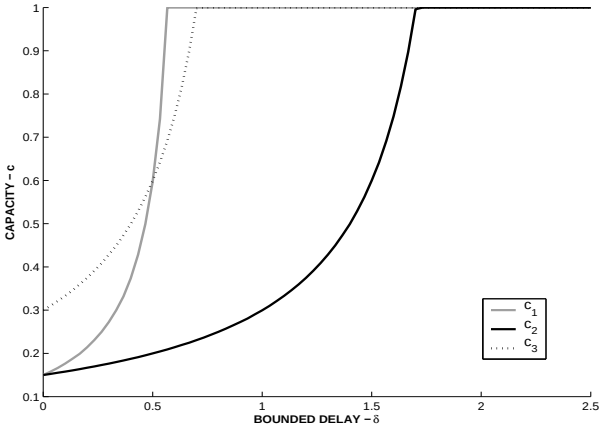

Figure 2. Capacity functions for Tab. 1

Component composition. In the formalism that we present in the next section the capacity function c_W represents a part of the interface of the component consisting of the task set W. In order to compose such components we need to compose resource assumptions in the form of such capacity functions. For this we again recall Thm. 3 [13], but now for the workload consisting of two bounded-delay tasks, $\{(c_1, \delta_1), (c_2, \delta_2)\}$. It follows from the theorem that this workload can be abstracted by the bounded-delay resource (c, δ), where $c = c_1 + c_2$ and $\delta = \min\{\delta_1, \delta_2\}$. This equation shows how to compute capacity functions for composite components: if two components (workloads) W_1 and W_2 are specified with their respective capacity functions c_{W_1} and c_{W_2}, then the sum c_{W_1}+c_{W_2} of the two functions ensures the schedulability of the composition. That is why, in our interface algebra (Sec. 3.2), when we perform component composition we add the corresponding capacity functions. Note that such an operation is associative. The task group composition, which we previously explained, does not have the associativity property.

3 Real-Time Interfaces

In Sec. 3.1 we motivate the assume-guarantee principle of the formalism, and we introduce interface predicates. In

Sec. 3.2 we formally define interfaces and prove an important proposition about interface refinement.

3.1 Informal Description

Let a component implement a single task sequence, i.e., let it have a single input port i and a single output port o. An interface is a constraint on the environment consisting of an input assumption and an output guarantee [2, 3]. Even though inputs (resp. outputs) of an actual component are task sequence request (resp. completion) events, in our formalism these events are abstracted by arrival rate functions. So, the values of the interface input and output ports are arrival rate functions. Let $\mathbb{A}$ be the set of all *arrival rate functions*, i.e., the set of all monotonically increasing functions a: $\mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$. An interface assumption may be that the input arrival rate $i \in \mathbb{A}$ is bounded by a given function $a \in \mathbb{A}$, i.e., $i \leq a$. Given the maximal delay $d \in \mathbb{R}_{\geq 0}$ of the component, for the arrival rate function of the output o we have the output guarantee that $o(t) \leq i(t + d)$ for all $t \in \mathbb{R}_{\geq 0}$. This inequality holds because, if the delay is at most d, then for all input requests in an interval of $t + d$ units, the outputs are produced in an interval of at least t units. Let i^d be the function defined as $i^d(t) = i(t + d)$.

A more expressive interface includes also a measure of resource consumption. We assume that such an interface contains an input port r, whose value is the capacity function of the component. Let $\mathbb{C}$ be the set of all *capacity functions*, i.e., the set of all monotonically increasing functions c: $\mathbb{R}_{\geq 0} \rightarrow [0, 1]$. The resource capacity assumption is $r \geq c$. Formally, the input predicate of the interface is $r \geq c \wedge i \leq a$, and the output predicate is $o \leq i^d$. This interface asserts that "if the environment provides a capacity larger than c, and input requests are upper bounded by a, then the component produces outputs with a delay smaller than d." Fig. 3 graphically represents an interface of a component that implements a sequence π_1 consisting of a single task τ_1. The interface is defined as a triple (a_1, d_1, c_1).

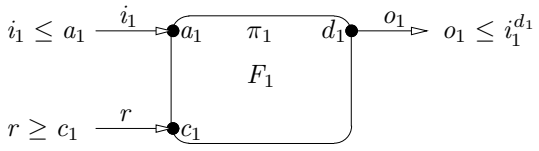

Figure 3. Interface for single task sequence

Let a component implement $n \in \mathbb{N}_{>0}$ task sequences through pairs (i_j, o_j) of input-output ports $(j = 1, \ldots, n)$. The interface of such a component bounds the arrival rate functions a_j of i_j and the delays d_j of o_j, for each $j = 1, \ldots, n$. Such a workload is still to be executed on a single resource partition whose capacity function c can be computed by task composition, as described in Sec. 2.2. We write F_W for the interface obtained by the composition of the tasks in the set W. Fig. 4 shows an interface with two single-task sequences and the corresponding predicates.

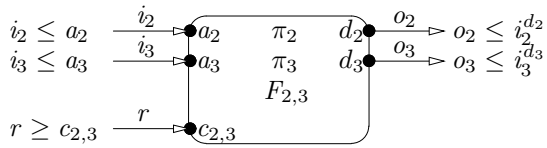

Figure 4. Interface for multiple sequences

We introduce two operations to construct more complex from simpler interfaces. The *composition* operation puts together the input and output ports of the two interfaces, and adds their resource capacity functions. The interface composition is defined if the sum of the two capacity functions is not larger than the constant function 1. Fig. 5 shows the interface resulting by composing the interface from Fig. 3 with the interface from Fig. 4. Thus, the interface describes three single-task sequences, $\pi_1 = \tau_1$, $\pi_2 = \tau_2$, and $\pi_3 = \tau_3$.

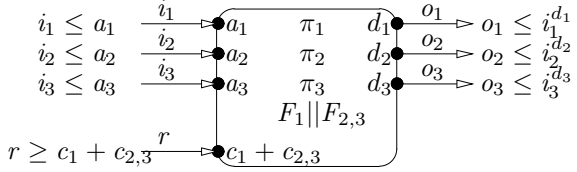

Figure 5. Interface composition

The *connection* operation connects the tasks of an interface into new sequences. This operation extends the set of interface task sequences, and therefore all input-output ports remain in the interface. Fig. 6 shows the interface from Fig. 5 after the two-task sequence $\pi_{12} = \tau_1\tau_2$ is created using the connection operation. The figure shows that the resource capacity assumption is not changed by the connection operation, and that the delay guarantee of a new sequence is computed as the sum of delays for the individual tasks of the sequence. The input assumptions of the resulting interface describe the most general constraints on the arrival rates for the extended set of task sequences. In particular, there is a constraint for each task that occurs in a sequence of the interface, namely, $i_1 + i_{12} \leq a_1$, $i_2 + i_{12}^{d_1} \leq a_2$, and $i_3 \leq a_3$. For instance, the rate of requests for task sequences that contain τ_2, i.e., the sum of the arrival rate functions i_2 and i_{12} (delayed by d_1), is bounded by a_2.

The input assumption of a newly created sequence π cannot be represented in a simple form $i_\pi \leq a_\pi$. To illustrate

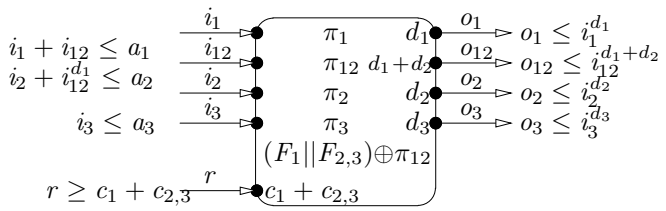

Figure 6. Interface connection

$S = \{\pi_{13}, \pi_{23}\}$,
$a_1(t) = 1 + t/2, a_2(t) = 1 + t/3, a_3 = a_1^{d_1} + a_2^{d_2}$,
$d_1 = 2/3, d_2 = 2, d_3 = 1$

F	F_a	F_b	F_c
Expr.	$F_{1,2,3}\oplus S$	$(F_{1,2}\|\|F_3)\oplus S$	$(F_1\|\|F_2\|\|F_3)\oplus S$
c_F	c_a	c_b	c_c
a_F	(a_1, a_2, a_3)		
d_F	(d_1, d_2, d_3)		

Table 2. Interface refinement

this, we consider bursty arrival rate functions for a two-task sequence $\pi_{12} = \tau_1\tau_2$ added by connection. We assume that τ_1 is specified with arrival rate function $a_1(t) = \sigma_1 + \rho_1 \cdot t$ and delay d_1, and τ_2 with $a_2(t) = \sigma_2 + \rho_2 \cdot t$ and d_2. If we assume the input arrival rate functions for the single-task sequences τ_1 and τ_2 to be 0, i.e., $i_1 = 0$ and $i_2 = 0$, then for the input arrival rate function i_{12} of the two-task sequence π_{12}, we have the constraints $i_{12} \le a_1$ and $i_{12}^{d_1} \le a_2$. If $i_{12}(t) = \sigma + \rho \cdot t$, then this is equivalent to $\sigma + \rho \cdot t \le \sigma_1 + \rho_1 \cdot t$, and $\sigma + \rho \cdot (t + d_1) \le \sigma_2 + \rho_2 \cdot t$. The values of σ and ρ that satisfy the two constraints are shown as a shaded area in the rightmost graph of Fig. 7. There is a trade-off between parameters σ and ρ, and this area cannot be specified in the $i_{12} \le a_{12}$ form.

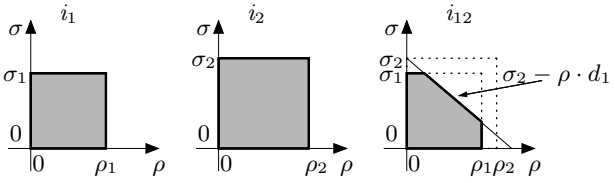

Figure 7. Bursty function parameters for i_1, i_2, and i_{12}

If the connection operation adds the three-task sequence $\pi_{123} = \tau_1\tau_2\tau_3$ to the interface $F_1\|F_{2,3}$, the resulting interface $(F_1\|F_{2,3})\oplus\pi_{123}$ contains an input and output port for each of the sequences π_1, π_2, π_3, and π_{123}. The input assumptions are $i_1 + i_{123} \le a_1$, $i_2 + i_{123}^{d_1} \le a_2$, and $i_3 + i_{123}^{d_1+d_2} \le a_3$. In the general case, the connection operation is defined as adding a set of sequences, each of which does not contain a cycle of tasks. For instance, the interface $(F_1\|F_{2,3})\oplus\{\pi_{12}, \pi_{21}\}$ consists of the input assumptions $i_1 + i_{12} + i_{21}^{d_2} \le a_1$, $i_2 + i_{12}^{d_1} + i_{21} \le a_2$, and $i_3 \le a_3$.

We also define a *refinement* relation between a more abstract and a more refined interface description. A more refined interface makes weaker input assumptions and stronger output guarantees than a more abstract interface. In that way, a refined description can always be substituted for an abstract one. From the expressions of interface input and output predicates it follows that an interface can be refined by either decreasing the capacity function, or increasing the arrival rate function for a task, or decreasing the delay of a task. Note that in Fig. 7 increasing σ_1, ρ_1, σ_2 or ρ_2, or decreasing d_1, makes the shaded areas larger, i.e., the constraints on inputs i_1, i_2, and i_{12} become weaker.

We first show examples of interface refinement through modification of the capacity function, while keeping other interface parameters constant. Fig. 8 shows the three interfaces F_a, F_b, and F_c of the component depicted in Fig. 1. The task composition operation (Sec. 2.2) is shown with rounded rectangles, and the interface composition with dashed rectangles. The interface expressions and parameters are given in Tab. 2. It is assumed that only the task sequences $\tau_1\tau_3$ and $\tau_2\tau_3$, and not the sequences τ_i for $i = 1, 2, 3$, are implemented by the component. Therefore, we assume that $a_3 = a_1^{d_1} + a_2^{d_2}$. All three interfaces have the same task arrival rate functions (a_1, a_2, a_3) and delays (d_1, d_2, d_3). However, the corresponding capacity functions c_a, c_b, and c_c are different because task composition is performed on different task sets. The interface F_a, for which task composition is applied on the entire task set, is characterized by a smaller capacity function than the interface F_c, for which each task is considered as a separate component. Fig. 9 shows that $c_a \le c_b \le c_c$. Therefore, we have that F_a refines F_b, F_b refines F_c, and F_a refines F_c.

For some interfaces it is possible to increase the arrival rate functions or decrease the delays of the interface task sequences, while keeping the resource capacity function constant. In some cases it is even possible to add a new task sequence to the component without affecting the capacity function. For instance, let $\tau_1 = (a_1, d_1, e_1) = (1 + t/2, 1, 0.2)$ and $\tau_2 = (a_2, d_2, e_2) = (1 + t/2, 2, 0.2)$, i.e., let the two tasks differ only in the delay. It can be shown that the capacity functions c_1 and $c_{1,2}$ for the independent task sets $\{\tau_1\}$ and $\{\tau_1, \tau_2\}$ are equal, i.e., $c_1 = c_{1,2}$. This is a consequence of the small delay requirement d_1. It follows that $F_{1,2}$ refines F_1. As explained in the following subsection, the definition of refinement allows a larger number of ports in the refined interface.

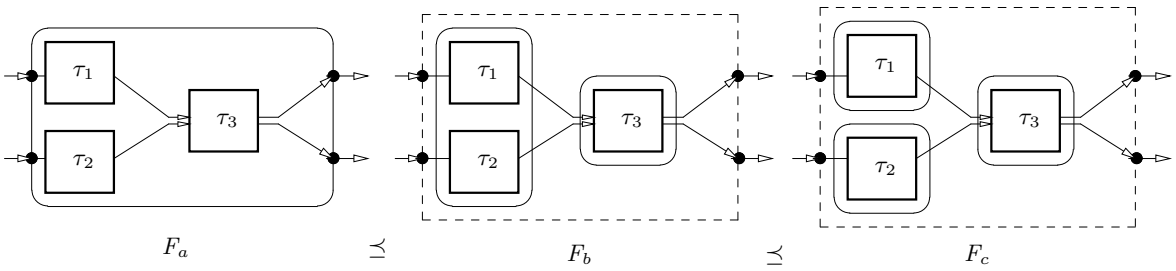

Figure 8. $F_a = F_{1,2,3} \oplus \{\pi_{13}, \pi_{23}\}$, $F_b = (F_{1,2} \| F_3) \oplus \{\pi_{13}, \pi_{23}\}$, $F_c = (F_1 \| F_2 \| F_3) \oplus \{\pi_{13}, \pi_{23}\}$

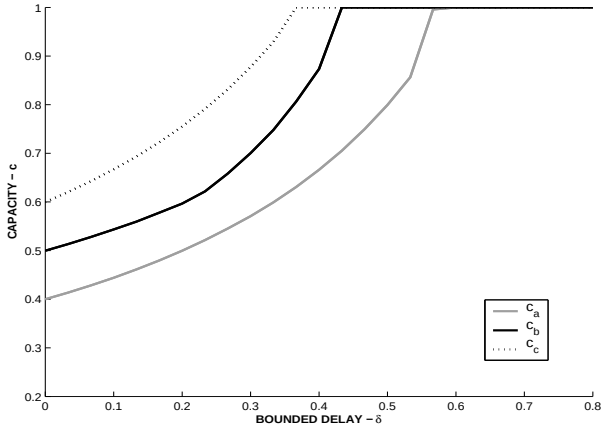

Figure 9. Capacity functions for Tab. 2

3.2 Interface Algebra

Let T be a set of tasks. A *task sequence* $\pi = \tau_1 \tau_2 \dots \tau_k$ is a finite sequence of different tasks $\tau_j \in T$, i.e., for all $1 \leq i < j \leq k$ we have $\tau_i \neq \tau_j$, and τ_i has to complete before τ_j starts execution.

An *interface* $F = (S_F, T_F^+, A_F, D_F, c_F)$ consists of:

- A set S_F of task sequences, and a set $T_F^+ \subseteq T$ of *available* tasks. The set T_F^+ contains the tasks that are available to the implementation of the interface F and its refinements. Let the set T_F of tasks contain all tasks that occur in the sequences in S_F, i.e., $T_F = \{\tau \in \pi \mid \pi \in S_F\}$. We require that (1) $T_F \subseteq T_F^+$ and (2) for every task $\tau \in T_F$, the sequence with the single task τ is contained in S_F.

  For each task sequence $\pi \in S_F$ there exist an input port i_π and an output port o_π. Let $I_F = \{i_\pi \mid \pi \in S_F\} \cup \{r\}$, $O_F = \{o_\pi \mid \pi \in S_F\}$, and $P_F = I_F \cup O_F$. The type of a port $x \in P_F$ is $\mathbb{A}$ (the set of arrival rate functions) if $x \neq r$, and $\mathbb{C}$ (the set of capacity functions) if $x = r$. A *valuation* v of a set P_F of ports is a function that maps each port $x \in P_F$ to a value $v(x)$ in the type of the port x.

- A function A_F that maps each task $\tau \in T_F$ to an arrival rate function $A_F(\tau) \in \mathbb{A}$, and a function D_F that maps each task $\tau \in T_F$ to a delay $D_F(\tau) \in \mathbb{R}_{\geq 0}$.

  Given a task sequence $\pi \in S_F$, let $D_F(\pi)$ be the sum of the delays of its tasks, i.e., $D_F(\pi) = \sum_{\tau \in \pi} D_F(\tau)$. By definition, for the empty task sequence ϵ, we have $D_F(\epsilon) = 0$.

- A capacity function $c_F \in \mathbb{C}$.

The *input predicate* (input assumption) $\phi_F^I = \phi^I(S_F, A_F, D_F, c_F)$ over the input ports I_F is defined to be

$$\phi_F^I = (r \geq c_F) \wedge \bigwedge_{\tau \in T_F} \Big(\sum_{\pi = \pi_1 \cdot \tau \cdot \pi_2 \in S_F} i_\pi^{D_F(\pi_1)} \leq A_F(\tau) \Big). \tag{1}$$

The *output predicate* (output guarantee) $\phi_F^O = \phi^O(S_F, D_F)$ over P_F is defined to be

$$\phi_F^O = \bigwedge_{\pi \in S_F} (o_\pi \leq i_\pi^{D_F(\pi)}). \tag{2}$$

The interface algebra for real-time components consists of:

- A partial binary function called *composition*, mapping two interfaces F and G to an interface $F \| G$. The composition $F \| G$ is defined if $T_F^+ \cap T_G^+ = \emptyset$, and if not $c_F + c_G > 1$, i.e., if $(c_F + c_G)(0) \leq 1$. If $F \| G$ is defined, then $S_{F\|G} = S_F \cup S_G$, $T_{F\|G}^+ = T_F^+ \cup T_G^+$, and $c_{F\|G} = \texttt{min}\{c_F + c_G, 1\}$. In addition, $A_{F\|G}(\tau) = A_F(\tau)$ if $\tau \in T_F$, and $A_{F\|G}(\tau) = A_G(\tau)$ if $\tau \in T_G$. Similarly, $D_{F\|G}(\tau) = D_F(\tau)$ if $\tau \in T_F$, and $D_{F\|G}(\tau) = D_G(\tau)$ if $\tau \in T_G$.

  The composition operation is commutative and associative. In particular, for all interfaces F, G, and H, if $(F \| G) \| H$ is defined, then $F \| (G \| H)$ is defined, and $(F \| G) \| H = F \| (G \| H)$.

- A partial binary function called *connection*, mapping an interface F and a set S of task sequences to an interface $F \oplus S$. The connection $F \oplus S$ is defined if T_F contains all tasks in the sequences in S, i.e., if for all sequences π of S, every task τ of π is also an element of T_F. If $F \oplus S$ is defined, then $S_{F \oplus S} = S_F \cup S$, $T^+_{F \oplus S} = T^+_F$, $A_{F \oplus S} = A_F$, $D_{F \oplus S} = D_F$, and $c_{F \oplus S} = c_F$.

  For all interfaces F and all sets S_1 and S_2 of task sequences, if $(F \oplus S_1) \oplus S_2$ is defined, then $(F \oplus S_2) \oplus S_1$ is defined, and $(F \oplus S_1) \oplus S_2 = F \oplus (S_1 \cup S_2) = (F \oplus S_2) \oplus S_1$. In addition, for all interfaces G, if $(F \| G) \oplus S_1$ and $F \oplus S_1$ are defined, then $(F \oplus S_1) \| G$ is defined, and $(F \| G) \oplus S_1 = (F \oplus S_1) \| G$.

- A binary relation $\preceq$ between interfaces, called *refinement*. An interface F' *refines* an interface F if (a) $S_{F'} \supseteq S_F$, (b) $T^+_{F'} \subseteq T^+_F$, and (c) for each valuation v_i of I_F and each valuation v'_o of $O_{F'}$, there exists a valuation v'_i of $I_{F'}$ such that $v_i = v'_i$ on I_F, and both implications $\phi^I_F \Rightarrow \phi^I_{F'}$ and $\phi^O_{F'} \Rightarrow \phi^O_F$ are valid.

  The relation $\preceq$ is reflexive and transitive.

One way to refine an interface is to apply the connection operation. Formally, if $F' = F \oplus S$ is defined, then $F' \preceq F$. This statement is true because the refinement condition (a) is satisfied since $S_{F'} = S_{F \oplus S} = S_F \cup S \supseteq S_F$, and (b) since $T^+_{F'} = T^+_{F \oplus S} = T^+_F$. If in the refinement condition (c) we take $v'_i(x) = 0$ for each $x \in I_{F'} \setminus I_F$ (i.e., the arrival rate functions for all sequences from $S \setminus S_F$ are 0), we have $\phi^I_{F'} = \phi^I_F$ and $\phi^O_{F'} = \phi^O_F$, and therefore $F' \preceq F$.

As informally discussed in Sec. 3.1, an interface $F = (S_F, T^+_F, A_F, D_F, c_F)$ can also be refined by increasing the arrival rate function for a task, or decreasing the delay of a task, or decreasing the capacity function. In the first case, if $F' = (S_F, T^+_F, A_{F'}, D_F, c_F)$ and if $A_{F'}(\tau) \geq A_F(\tau)$ for each $\tau \in T_F$, then $F' \preceq F$. In this case the refinement condition (c) is satisfied because for every valuation of P_F, from Eq. 1 it follows $\phi^I_F = \phi^I(S_F, A_F, D_F, c_F) \Rightarrow \phi^I(S_F, A_{F'}, D_F, c_F) = \phi^I_{F'}$, and from Eq. 2 it follows $\phi^O_{F'} = \phi^O(S_F, D_F) = \phi^O_F$. Similar arguments can be presented in other two cases. Namely, if $F' = (S_F, T^+_F, A_F, D_{F'}, c_F)$ and $D_{F'}(\tau) \leq D_F(\tau)$ for each $\tau \in T_F$, then $F' \preceq F$. Also, if $F' = (S_F, T^+_F, A_F, D_F, c_{F'})$ and $c_{F'} \leq c_F$, then $F' \preceq F$.

The following proposition justifies the independent implementability property discussed in Sec. 4.2.

Proposition 1 *For all interfaces F, F', and G, and all sets S of task sequences,*

1. *If $F \| G$ is defined and $F' \preceq F$, then $F' \| G$ is defined and $F' \| G \preceq F \| G$.*
2. *If $F \oplus S$ is defined and $F' \preceq F$, then $F' \oplus S$ is defined and $F' \oplus S \preceq F \oplus S$.*

Proof. We sketch only the proof for part 2. The proof for part 1 is simpler, and similar. To simplify notation, we first introduce the following predicates: $\phi^r_F = (r \geq c_F)$, $\phi^i_F = \bigwedge_{\tau \in T_F} (\sum_{\pi = \pi_1 \cdot \tau \cdot \pi_2 \in S_F} i_\pi^{D_F(\pi_1)} \leq A_F(\tau))$, and $\phi^O_{F,S} = \bigwedge_{\pi \in S} (o_\pi \leq i_\pi^{D_F(\pi)})$. Note that $\phi^I_F = (\phi^r_F \wedge \phi^i_F)$.

If $F \oplus S$ is defined, then T_F contains all tasks in all sequences of S. Since $T_{F'} \supseteq T_F$, the same is true for $T_{F'}$, and therefore the interface $F' \oplus S$ is defined. From the definitions of interface connection and refinement, it follows that $S_{F' \oplus S} \supseteq S_{F \oplus S}$ and $T^+_{F' \oplus S} \subseteq T^+_{F \oplus S}$. If $F' \preceq F$, then for each valuation on I_F, there exists a valuation on I'_F such that the implication $\phi^I_F \Rightarrow \phi^I_{F'}$ is valid, because both $\phi^r_F \Rightarrow \phi^r_{F'}$ and $\phi^i_F \Rightarrow \phi^i_{F'}$ are valid. If $\phi^r_F \Rightarrow \phi^r_{F'}$ is valid for each value of $r \in I_F$, then $c_F \geq c_{F'}$, and therefore $c_{F \oplus S} = c_F \geq c_{F'} = c_{F' \oplus S}$. Consequently, for each r, the implication $\phi^r_{F \oplus S} \Rightarrow \phi^r_{F' \oplus S}$ is valid. If $F' \preceq F$, then also the implication $\phi^O_{F'} \Rightarrow \phi^O_F$ is valid, because for each sequence π of S_F, the predicate $o_\pi \leq i_\pi^{D_{F'}(\pi)}$ implies the predicate $o_\pi \leq i_\pi^{D_F(\pi)}$. It follows that $i_\pi^{D_{F'}(\pi)} \leq i_\pi^{D_F(\pi)}$, i.e., $D_{F'}(\pi) \leq D_F(\pi)$ for each sequence $\pi \in S_F$. Since the single-task sequence $\pi = \tau$ is in S_F for each $\tau \in T_F$, we have $D_{F'}(\tau) \leq D_F(\tau)$. Similarly, from $\phi^i_F \Rightarrow \phi^i_{F'}$, we obtain $A_{F'}(\tau) \geq A_F(\tau)$ for each $\tau \in T_F$. For a given $\tau \in T_{F \oplus S}$, if $\sum_{\pi = \pi_1 \cdot \tau \cdot \pi_2 \in S_F} i_\pi^{D_F(\pi_1)} \leq A_F(\tau)$, and if we take $i_\pi = 0$ for each $\pi \in S_{F'} \setminus (S_F \cup S)$, we obtain

$$\sum_{\pi \in S_{F' \oplus S}} i_\pi^{D_{F'}(\pi_1)} \leq \sum_{\pi \in S_{F \oplus S}} i_\pi^{D_F(\pi_1)} \leq A_F(\tau) \leq A_{F'}(\tau).$$

Consequently, $\phi^i_{F \oplus S} \Rightarrow \phi^i_{F' \oplus S}$, and therefore $\phi^I_{F \oplus S} \Rightarrow \phi^I_{F' \oplus S}$. Since $D_{F'}(\pi) \leq D_F(\pi)$ also for each sequence $\pi \in S$, it follows that the implication $\phi^O_{F',S} \Rightarrow \phi^O_{F,S}$ is valid. If both the predicate $\phi^O_{F' \oplus S} = (\phi^O_{F'} \wedge \phi^O_{F',S})$ and the implication $\phi^O_{F'} \Rightarrow \phi^O_F$ are valid, then $\phi^O_{F \oplus S} = \phi^O_F \wedge \phi^O_{F,S}$ is valid, and hence $\phi^O_{F' \oplus S} \Rightarrow \phi^O_{F \oplus S}$ is valid. □

Let $f(F_1, ..., F_k, S_1, ..., S_l)$ be the interface computed by applying finitely many composition and connection operations on the interfaces $F_1, ..., F_k$ and task sequences $S_1, ..., S_l$. From Prop. 1 it follows that this interface can be refined through independent refinement of the interfaces F_j, i.e., if $F'_j \preceq F_j$ for $j = 1, ..., k$, then $f(F'_1, ..., F'_k, S_1, ..., S_l)$ is defined and $f(F'_1, ..., F'_k, S_1, ..., S_l) \preceq f(F_1, ..., F_k, S_1, ..., S_l)$.

4 Real-Time Component-Based Design

We illustrate on a real-time system example how the interface formalism supports incremental design and independent implementability. In this section we allow for a more

general form of an interface than the one formally presented in Sec. 3. Namely, we assume that a component may provide multiple levels of service for the task sequences it implements. At any time the component may change from the current to any other level of service. This is a simple extension towards a stateful interface formalism. We require that at all levels of service the component implements the same set of task sequences. Let F denote a *single-level* interface as defined in Sec. 3. Formally, a *multi-level* interface $\mathcal{F}$ is a finite set of single-level interfaces such that if $F_1, F_2 \in \mathcal{F}$, then $S_{F_1} = S_{F_2}$. In particular, a multi-level interface $\mathcal{F}$ is specified with tuples (A_F, D_F, c_F) for each single-level interface $F \in \mathcal{F}$. Let $m_{\mathcal{F}}$ be the number of service levels provided by $\mathcal{F}$, i.e., the number of elements of the set $\mathcal{F}$.

Two multi-level interfaces $\mathcal{F}$ and $\mathcal{G}$ are compatible if there exist a level of service of $\mathcal{F}$ and a level of service of $\mathcal{G}$ that are compatible, i.e., if there exist $F \in \mathcal{F}$ and $G \in \mathcal{G}$ such that $F \| G$ is defined. In that case, the composition $\mathcal{F} \| \mathcal{G}$ is defined and $\mathcal{F} \| \mathcal{G} = \{F \| G \mid F \in \mathcal{F}\ ,\ G \in \mathcal{G}\ ,\ F \| G \text{ is defined}\}$. Therefore, in the worst case, checking compatibility requires $O(m_{\mathcal{F}} \cdot m_{\mathcal{G}})$ time. The connection operator on $\mathcal{F}$ applies the operator on each single-level interface in $\mathcal{F}$, i.e., $\mathcal{F} \oplus S = \{F \oplus S \mid F \in \mathcal{F}\}$. A multi-level interface $\mathcal{F}'$ refines $\mathcal{F}$, if $\mathcal{F}'$ can be substituted for $\mathcal{F}$ in each level of service of $\mathcal{F}$, i.e., if for each $F \in \mathcal{F}$, there exists $F' \in \mathcal{F}'$ such that $F' \preceq F$.

4.1 Incremental Design

Since the interface composition is associative, the order in which we compose components makes no difference. Moreover, this means that compatibility can be checked even before all interfaces are fully specified, i.e., before the system becomes closed. Formally, we can check whether n interfaces are compatible, i.e., whether $\mathcal{F}_1 \| \ldots \| \mathcal{F}_{n-1} \| \mathcal{F}_n$ is defined, by constructing $(\mathcal{F}_1 \| \ldots \| \mathcal{F}_{i-1}) \| \mathcal{F}_i$ for $i = 1, \ldots, n$. The computational complexity of this operation is typically less than $m_{\mathcal{F}_1} \cdot \ldots \cdot m_{\mathcal{F}_n}$ because incompatible levels of service are eliminated as soon as possible. This procedure can be further improved by composing interfaces in a tree-like order, rather than in a linear order.

We demonstrate the efficiency of incremental design on a real-time robotic application adapted from [6]. In this real-time system, there are five task sequences π_i $(i = 1, \ldots, 5)$ using a total of 13 tasks, as shown in the underlying graph in Fig. 10. Let $S_4 = \{\pi_1, \pi_2, \pi_3, \pi_4\}$ and $S_5 = S_4 \cup \{\pi_5\}$. The system is planned for three levels of service: 80%, 100%, and 120% of the nominal arrival rates for π_i, which are given in the ρ row of Tab. 3. The execution times of all tasks are also part of the system specification. The other task data in Tab. 3 are computed assuming that a sequence delay is inversely proportional to its rate. Similar tables are computed for the other two levels of service.

π	π_1			π_2			π_3		π_4			π_5	
τ	τ_{11}	τ_{12}	τ_{13}	τ_{21}	τ_{22}	τ_{23}	τ_{31}	τ_{32}	τ_{41}	τ_{42}	τ_{43}	τ_{51}	τ_{52}
ρ	0.015			0.031			0.124		0.062			0.156	
e	0.2	1.2	1.0	1.0	2.0	0.3	0.8	1.2	1.0	0.5	0.5	0.1	0.5
d	10.48	32.05	21.56	5.79	16.37	9.88	2.30	5.71	3.33	7.51	5.17	1.55	4.86
σ	1	1.16	1.64	1	1.15	1.59	1	1.19	1	1.14	1.58	1	1.23

Table 3. Task data for nominal arrival rates

Fig. 10 and 12 show two different component decompositions of the system. Let the system be composed of the components A, B, and C as shown in Fig. 10. The capacity functions of the corresponding interfaces $\mathcal{F}_A$, $\mathcal{F}_B$, and $\mathcal{F}_C$ are computed by the task composition procedure for each of the three levels of service ($m_{\mathcal{F}_A} = m_{\mathcal{F}_B} = m_{\mathcal{F}_C} = 3$). The results of checking for interface compatibility are shown in Fig. 11. Each table row represents a combination of levels of components for which the composition is defined. Instead of showing entire capacity functions, in the last two columns of the table we characterize the functions with two numbers: $c(0)$ is the resource capacity at delay 0, and δ_1 is the delay at which the capacity has to be 1. The interface $(\mathcal{F}_A \| \mathcal{F}_B) \oplus S_4$ consists of 7 levels of service, since 2 are eliminated due to incompatibility. Similarly, the interface for the entire system consists of 10, and not $3^3 = 27$, levels of service.

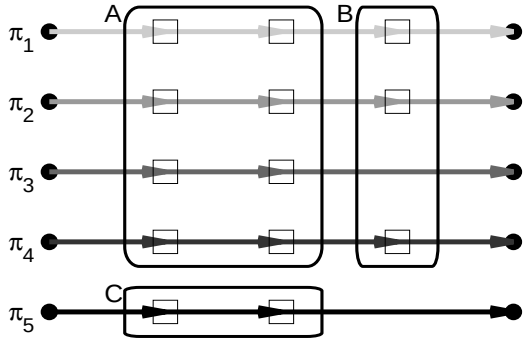

Figure 10. $(\mathcal{F}_A \| \mathcal{F}_B \| \mathcal{F}_C) \oplus S_5$

If the system is composed of the components a, b, and C, as shown in Fig. 12, the resulting interfaces $(\mathcal{F}_a \| \mathcal{F}_b) \oplus S_4$ and $(\mathcal{F}_a \| \mathcal{F}_b \| \mathcal{F}_C) \oplus S_5$ are shown in Fig. 13. The table shows that with this composition only three combinations of service levels are attainable, even though the properties of the arrival sequences and tasks are the same as for Fig. 10. This confirms that, although interface composition is associative, task composition is not. In particular, even though $((\mathcal{F}_A \| \mathcal{F}_B) \| \mathcal{F}_C) \oplus S_5 = (\mathcal{F}_A \| (\mathcal{F}_B \| \mathcal{F}_C)) \oplus S_5$, and $((\mathcal{F}_a \| \mathcal{F}_b) \| \mathcal{F}_C) \oplus S_5 = (\mathcal{F}_a \| (\mathcal{F}_b \| \mathcal{F}_C)) \oplus S_5$, the interfaces $(\mathcal{F}_A \| \mathcal{F}_B \| \mathcal{F}_C) \oplus S_5$ and $(\mathcal{F}_a \| \mathcal{F}_b \| \mathcal{F}_C) \oplus S_5$ are not equivalent.

$(\mathcal{F}_A \| \mathcal{F}_B) \oplus S_4$				
j	$\mathcal{F}_A$	$\mathcal{F}_B$	$c(0)$	δ_1
1	80	80	0.71	1.80
2	80	100	0.75	1.60
3	80	120	0.78	1.30
4	100	80	0.86	0.80
5	100	100	0.89	0.60
6	100	120	0.92	0.40
7	120	80	0.98	0.00

$(\mathcal{F}_A \| \mathcal{F}_B \| \mathcal{F}_C) \oplus S_5$					
j	$\mathcal{F}_A$	$\mathcal{F}_B$	$\mathcal{F}_C$	$c(0)$	δ_1
1	80	80	80	0.83	1.10
2	80	80	100	0.86	0.90
3	80	80	120	0.89	0.70
4	80	100	80	0.86	0.90
5	80	100	100	0.89	0.70
6	80	100	120	0.92	0.40
7	80	120	80	0.89	0.60
8	80	120	100	0.92	0.40
9	80	120	120	0.95	0.30
10	100	80	80	0.97	0.10

Figure 11. Levels of service for $(\mathcal{F}_A \| \mathcal{F}_B) \oplus S_4$ and $(\mathcal{F}_A \| \mathcal{F}_B \| \mathcal{F}_C) \oplus S_5$

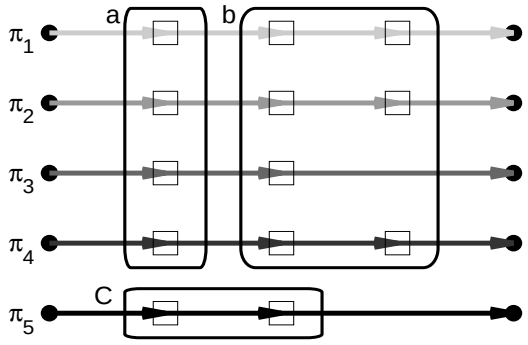

Figure 12. $(\mathcal{F}_a \| \mathcal{F}_b \| \mathcal{F}_C) \oplus S_5$

4.2 Independent Implementability

The formalism presented in Sec. 3 enables compositional refinement, i.e., it enables the independent refinement of component interfaces by component implementations. From Prop. 1 it follows that in order to refine a given composition of interfaces, it suffices to independently refine each interface and to compose the obtained refinements. The higher efficiency of such a procedure lies in the fact that now refinement checks involve smaller interfaces. In that way, a single complex problem is reduced to multiple simpler problems.

Suppose that the robotic application from the previous subsection is planned for the task sequence arrival rates that are 80% of the nominal rates, and that it is given a full resource. Namely, assume that the specification for the application is given as a single-level interface F, with the

$(\mathcal{F}_a \| \mathcal{F}_b) \oplus S_4$				
j	$\mathcal{F}_a$	$\mathcal{F}_b$	$c(0)$	δ_1
1	80	80	0.82	1.00
2	80	100	0.91	0.50
3	100	80	0.91	0.40
4	120	80	0.99	0.00

$(\mathcal{F}_a \| \mathcal{F}_b \| \mathcal{F}_C) \oplus S_5$					
j	$\mathcal{F}_a$	$\mathcal{F}_b$	$\mathcal{F}_C$	$c(0)$	δ_1
1	80	80	80	0.94	0.30
2	80	80	100	0.97	0.20
3	80	80	120	1.00	0.00

Figure 13. Levels of service for $(\mathcal{F}_a \| \mathcal{F}_b) \oplus S_4$ and $(\mathcal{F}_a \| \mathcal{F}_b \| \mathcal{F}_C) \oplus S_5$

task arrival rate functions A_F and delays D_F computed as in Tab. 3 for the 80% of rates, and with the capacity function c_F such that $c_F(0) = 1$. The design goal is to implement the system as a composition of real-time components that refine the specification interface F. To that purpose the specification is represented as the composition $F = (F_a \| F_b \| F_C) \oplus S_5$ from Fig. 12 of the three interfaces F_a, F_b, and F_C, aimed for independent implementation. In addition, assume that an initial analysis shows that the resource should be distributed 45% to the interface F_a, 40% to the interface F_b, and 15% to the interface F_C. Hence, $c_{F_a}(0) = 0.45$, $c_{F_b}(0) = 0.4$, and $c_{F_C}(0) = 0.15$.

When in the implementation process the wcet's become known, the task composition procedure can be used to check for the actual resource requirements of the components. Moreover, the three interfaces can be refined by increasing the arrival rates of some task sequences. In particular, it can be shown that the interface F'_C, which is obtained from the interface F_C by increasing the rate of the task sequence π_5 to 102% of the nominal rate, still satisfies $c_{F'_C}(0) = 0.15$. Therefore, it follows $F'_C \preceq F_C$. Similarly, 82% of the nominal rate for π_4 in interface F'_a results in $c_{F'_a}(0) = 0.45$, and 110% of the nominal rate for π_4 in interface F'_b results in $c_{F'_b}(0) = 0.4$. Hence, $F'_a \preceq F_a$ and $F'_b \preceq F_b$. According to Prop. 1, the composition of interface refinements, $F' = (F'_a \| F'_b \| F'_C) \oplus S_5$, refines the original specification interface F. In particular, the refined interface F' allows for higher rates of the task sequences π_4 and π_5 than 80% of their nominal rates that is originally specified in F.

5 Conclusion

We showed how a group of tasks, each defined with an arrival rate, a delay, and a worst-case execution requirement, can be abstracted into a bounded-delay resource model. In order to use such abstracted components in a larger real-time system comprising of multiple task sequences, we introduced component interfaces. A formal in-

terface algebra allows for automatic procedures that help in component integration. We motivated and analyzed two properties of such a framework, incremental design and independent implementability. The composition with abstracted components inevitably incurs higher resource utilization and, therefore, the effectiveness of composition can be compromised. We leave the question of how tight the entire framework is for future work. Richer task models, such as the one in which the underlying task precedence graph is a DAG, are also worth investigating. In addition, interesting problems for future investigations arise when more complex, temporally adaptive interactions between real-time components require an automaton-based interface formalism.

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *EMSOFT*, pages 95–103. ACM, 2004.

[2] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120. ACM, 2001.

[3] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2001.

[4] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed interfaces. In *EMSOFT*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2002.

[5] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS*, pages 308–319. IEEE Computer Society, 1997.

[6] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1):13–28, 1994.

[7] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *ECRTS*, pages 151–158. IEEE Computer Society, 2003.

[8] A. K. Mok and A. X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. In *RTSS*, pages 129–138. IEEE Computer Society, 2001.

[9] A. K. Mok and A. X. Feng. A model of hierarchical real-time virtual resources. In *RTSS*, pages 26–35. IEEE Computer Society, 2002.

[10] A. K. Mok, A. X. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS*, pages 75–84. IEEE Computer Society, 2001.

[11] J. Regehr and J. A. Stankovic. Hls: A framework for composing soft real-time schedulers. In *RTSS*, pages 3–14. IEEE Computer Society, 2001.

[12] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS*, pages 2–13. IEEE Computer Society, 2003.

[13] I. Shin and I. Lee. Compositional real-time scheduling framework. In *RTSS*, pages 57–67. IEEE Computer Society, 2004.

[14] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *EMSOFT*, pages 80–89. ACM, 2005.

[15] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao. Real-time component-based systems. In *RTAS*, pages 428–437. IEEE Computer Society, 2005.