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 \((c; \delta)\), studied in [10, 8, 9], guarantees fraction \(c\) of the resource with at most \(\delta\) 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
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 \( \tau_j \) and \( \tau_{j+1} \) 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 \( \lceil a(t) \rceil \).

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 \( \mathcal{R} = (c, \delta) \) 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 \( \delta \) addresses the trade-off between high context switch costs (smaller \( \delta \)) and high task execution latencies (larger \( \delta \)).

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

\[
\text{sbf}_\mathcal{R}(t) = \text{sbf}_{(c,\delta)}(t) = \begin{cases} 
0 & \text{if } t \leq \delta, \\
\c(t - \delta) & \text{if } t > \delta.
\end{cases}
\]

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 \( \mathcal{R} \) be a bounded-delay resource model. We say that \( (W, \mathcal{R}, \text{EDF}) \) is schedulable if under every instance of allocations of the resource \( \mathcal{R} \), there exists a feasible EDF schedule for \( W \) [12]. If \( (W, \mathcal{R}, \text{EDF}) \) is schedulable, then the set of tasks \( W \) under the resource \( \mathcal{R} = (c, \delta) \) and the EDF scheduling algorithm can be abstracted as a single requirement \( (c, \delta) \), i.e., no global knowledge of task internals is necessary. A discussion on how to schedule several \( (c, \delta) \) 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 \( \tau_j \), 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 \( \text{dbf} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \) 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 \( \text{dbf}_W(t) = \sum_{\tau_j \in W} [t/p_j] \cdot e_j \).

For the case of periodic workloads \( W \), Thm. 1 in [13] gives a sufficient and necessary condition for schedulability: \( (W, \mathcal{R}, \text{EDF}) \) is schedulable if for all \( 0 < t \leq 2 \cdot \text{lcm}_W \), the maximal resource demand is no greater than the minimum resource supply, i.e., \( \text{dbf}_W(t) \leq \text{sbf}_{\mathcal{R}}(t) \). In this condition, \( \text{lcm}_W \) 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 \( \text{dbf}_W \) can be computed: \( (W, \mathcal{R}, \text{EDF}) \) is schedulable if for all \( t > 0 \), we have \( \text{dbf}_W(t) \leq \text{sbf}_{\mathcal{R}}(t) \).
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 $\tau_j$, 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 $\tau_j$. We note that there are at most $[a_j(t - d_j)]$ invocations of the task $\tau_j$ that are released and required to complete in a time interval of length $t$. Therefore, we have

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

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

Table 1. Temporal interface for Fig. 1

<table>
<thead>
<tr>
<th>$W$</th>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>$\rho$</td>
<td>1/2</td>
<td>1/3</td>
<td>1/2+1/3</td>
</tr>
<tr>
<td>$d$</td>
<td>2/3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$e$</td>
<td>0.1</td>
<td>0.3</td>
<td>0.1</td>
</tr>
</tbody>
</table>

We apply this result for the case of the aperiodic workload defined with arrival rate functions and delays for tasks. 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 $x \in X$.

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 $(c, \delta)$, 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.
functions

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 $a$ 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 $i \in a$ is bounded by a given function $a \in a$, 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 $c$ 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 $r \geq c \land i \leq a$, 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 $\pi_1$ consisting of a single task $\tau_1$. The interface is defined as a triple $(a_1, d_1, c)$. 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](image)

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](image)

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, $i_1 + i_2 \leq o_1$, $i_2 + i_3 \leq o_2$, and $i_3 \leq o_3$. For instance, the rate of requests for task sequences that contain $\tau_2$, i.e., the sum of the arrival rate functions $i_2$ and $i_2$ (delayed by $d_1$), is bounded by $o_2$.

![Figure 6. Interface after connection](image)
this, we consider bursty arrival rate functions for a two-task sequence \( \pi_{12} = \tau_1 \tau_2 \) added by connection. We assume that \( \tau_1 \) is specified with arrival rate function \( a_1(t) = \sigma_1 + \rho_1 \cdot t \) and delay \( d_1 \), and \( \tau_2 \) 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 \( \tau_1 \) and \( \tau_2 \) 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 \( \pi_{12} \), we have the constraints \( i_{12} \leq a_1 \) and \( i_{12}^{d_1} \leq a_2 \). If \( i_{12}(t) = \sigma + \rho \cdot t \), then this is equivalent to \( \sigma + \rho \cdot t \leq \sigma_1 + \rho_1 \cdot t, \) and \( \sigma + \rho \cdot (t + d_1) \leq \sigma_2 + \rho_2 \cdot t \). The values of \( \sigma \) and \( \rho \) 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 \( \sigma \) and \( \rho \), and this area cannot be specified in the \( i_{12} \leq a_{12} \) form.

If the connection operation adds the three-task sequence \( \pi_{123} = \tau_1 \tau_2 \tau_3 \) to the interface \( F_{1} \parallel F_{2,3} \), the resulting interface \( (F_{1} \parallel F_{2,3}) \oplus \pi_{123} \) contains an input and output port for each of the sequences \( \pi_1, \pi_2, \pi_3, \) and \( \pi_{123} \). The input assumptions are \( i_1 + i_{123} \leq a_1, i_2 + i_{123}^{d_1} \leq a_2, \) and \( i_3 + i_{123}^{d_1+d_2} \leq a_3 \). 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 \( (F_{1} \parallel F_{2,3}) \oplus \{ \pi_{12}, \pi_{21} \} \) consists of the input assumptions \( i_1 + i_{12} + i_{21}^{d_2} \leq a_1, i_2 + i_{12}^{d_1} + i_{21} \leq a_2, \) and \( i_3 \leq a_3 \).

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 \( \sigma_1, \sigma_2, \rho_1, \rho_2, \) or \( d_1 \), 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 \( \tau_1 \tau_2 \) and \( \tau_2 \tau_3 \), and not the sequences \( \tau_i \) for \( i = 1, 2, 3 \), are implemented by the component. Therefore, we assume that \( a_i = a_1^{d_1} + a_2^{d_2} \). 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 \( c_a \leq c_b \leq c_c \). 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.
is a finite sequence of different tasks
A set
Let
0.2
0.4
0.6
0.8
1
is contained in
S
starts execution.
For each task sequence
functions) if
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 \( \epsilon \), we have \( D_F(\epsilon) = 0 \).
• A capacity function \( c_F \in \mathbb{C} \).

The input predicate (input assumption) \( \phi^I_F = \phi^I(S_F, A_F, D_F, c_F) \) over the input ports \( I_F \) is defined to be
\[
\phi^I_F = (r \geq c_F) \land \bigwedge_{\pi \in S_F} \left( \sum_{\tau \in \pi} i^D_F(\tau) \leq A_F(\tau) \right).
\]
(1)
The output predicate (output guarantee) \( \phi^O_F = \phi^O(S_F, D_F) \) over \( P_F \) is defined to be
\[
\phi^O_F = \bigwedge_{\pi \in S_F} \left( o^d \leq i^D_F(\pi) \right).
\]
(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 \( F \parallel G \). The composition \( F \parallel G \) 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 \( F \parallel G \) is defined, then \( S_{F \parallel G} = S_F \cup S_G \), \( T_{F \parallel G}^+ = T_F^+ \cup T_G^+ \), and \( c_{F \parallel G} = \min\{c_F + c_G, 1\} \). In addition, \( A_{F\parallel G}(\tau) = A_F(\tau) \) if \( \tau \in T_F \), and \( A_{F\parallel G}(\tau) = A_G(\tau) \) if \( \tau \in T_G \). Similarly, \( D_{F\parallel G}(\tau) = D_F(\tau) \) if \( \tau \in T_F \), and \( D_{F\parallel G}(\tau) = D_G(\tau) \) if \( \tau \in T_G \). The composition operation is commutative and associative. In particular, for all interfaces \( F, G, \) and \( H \), if \( (F \parallel G) \parallel H \) is defined, then \( F \parallel (G \parallel H) \) is defined, and \( (F \parallel G) \parallel H = F \parallel ((G \parallel H)) \).
A binary relation $\succeq$ 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 $v_1$ of $I_F$ and each valuation $v'_1$ of $O_{F'}$, there exists a valuation $v'_1$ of $I_{F'}$ such that $v_1 = v'_1$ on $I_F$, and both implications $\phi_F \Rightarrow \phi_{F'}$ and $\phi_{F'} \Rightarrow \phi_F$ are valid.

The relation $\succeq$ 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 the refinement condition (c) we take $v'(x) = 0$ for each $x \in O_{F'} \setminus I_F$ (i.e., the arrival rate functions for all sequences from $S \setminus S_F$ are 0), we have $\phi_{F'} = \phi_F$ and $\phi_F \Rightarrow \phi_{F'}$, 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 $\phi_{F'} = \phi'(S_{F'}, T_{F'}^+, A_{F'}, D_{F'}, c_{F'}) \Rightarrow \phi'(S_F, T_F^+, A_F, D_F, c_F) = \phi_F$, and from Eq. 2 it follows $\phi_{F'}^0 = \phi_0(S_{F'}, T_{F'}^+, A_{F'}, D_{F'}, c_{F'}) = \phi_{F'}^0$. 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 $F \parallel G$ is defined and $F' \preceq F$, then $F' \parallel G$ is defined and $F' \parallel G \preceq F \parallel G$.

2. If $F \parallel S$ is defined and $F' \preceq F$, then $F' \parallel S$ is defined and $F' \parallel S \preceq F \parallel S$.

**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: $\phi_F = (r \geq c_F), \phi_{F'} = \bigwedge_{\tau \in T_F} (\sum_{i=1}^{n_{F/\tau}} i_{D_F(\pi_i)} \leq A_F(\tau)), \phi_{F',S} = \bigwedge_{\tau \in S} (o_\tau \leq i_{D_F(\pi)}).$ Note that $\phi_F = (\phi_F \land \phi_F')$.

If $F \parallel S$ is defined, then $T_F$ contains all tasks in all sequences of $S$. Since $T_{F'} \subseteq T_F$, the same is true for $T_{F'}$, and therefore the interface $F' \parallel S$ is defined. From the definitions of interface connection and refinement, it follows that $S_{F' \parallel S} \supseteq S_{F \parallel S}$ and $T_{F' \parallel S} \subseteq T_{F' \parallel S}$. If $F' \preceq F$, then for each valuation on $I_{F'}$, there exists a valuation on $I_F$ such that the implication $\phi_{F'} \Rightarrow \phi_F$ is valid, because both $\phi_{F'} \Rightarrow \phi_{F'}$ and $\phi_{F'} \Rightarrow \phi_F$ are valid. If $\phi_{F'} \Rightarrow \phi_F$ is valid for each value of $r \in I_F$, then $c_F \geq c_{F'}$, and therefore $c_{F' \parallel S} = c_F \geq c_{F' \parallel S} = c_{F' \parallel S}$. Consequently, for each $r$, the implication $\phi_{F' \parallel S} \Rightarrow \phi_{F' \parallel S}$ is valid. If $F' \preceq F$, then also the implication $\phi_F \Rightarrow \phi_{F'}$ is valid, because for each sequence $\pi$ of $S_{F'}$, the predicate $o_\pi \leq i_{D_{F'}(\pi)}$ implies the predicate $o_\pi \leq i_{D_F(\pi)}$. It follows that $i_{D_F(\pi)} \leq i_{D_{F'}(\pi)}$, 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 $\phi_{F'} \Rightarrow \phi_F$, we obtain $A_{F'}(\tau) \geq A_F(\tau)$ for each $\tau \in T_F$. For a given $\tau \in T_{F' \parallel S}$, if $\sum_{\pi=1}^n i_{D_F(\pi)} \leq A_F(\tau)$, and if we take $i_\pi = 0$ for each $\pi \in S_{F'} \setminus (S_F \cup S)$, we obtain

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

Consequently, $\phi_{F' \parallel S} \Rightarrow \phi_{F' \parallel S}$, and therefore $\phi_{F' \parallel S} \Rightarrow \phi_{F' \parallel S}$. Since $D_{F'}(\pi) \leq D_F(\pi)$ also for each sequence $\pi \in S$, it follows that the implication $\phi_{F' \parallel S} \Rightarrow \phi_{F' \parallel S}$ is valid. If both the predicate $\phi_{F' \parallel S} = (\phi_{F'} \land \phi_{F' \parallel S})$ and the implication $\phi_{F'} \Rightarrow \phi_{F'}$ are valid, then $\phi_{F' \parallel S} = \phi_{F'} \land \phi_{F' \parallel S}$ is valid, and hence $\phi_{F' \parallel S} \Rightarrow \phi_{F' \parallel S}$ 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 $F$ is a finite set of single-level interfaces such that if $F_1, F_2 \in F$, then $S_{F_1} = S_{F_2}$. In particular, a multi-level interface $F$ is specified with tuples $(A_F, D_F, c_F)$ for each single-level interface $F \in F$. Let $m_F$ be the number of service levels provided by $F$, i.e., the number of elements of the set $F$.

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

### 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 $F_1 \parallel \ldots \parallel F_n$ is defined, by constructing $(F_1 \parallel \ldots \parallel F_{i-1}) \parallel F_i$ for $i = 1, \ldots, n$. The computational complexity of this operation is typically less than $m_{F_1} \cdot \ldots \cdot m_{F_n}$ 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 $\pi_1 (i = 1, \ldots, 5)$ 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 $\pi_1$, which are given in the $\rho$ 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.

<table>
<thead>
<tr>
<th>$\pi_1$</th>
<th>$\pi_2$</th>
<th>$\pi_3$</th>
<th>$\pi_4$</th>
<th>$\pi_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>$\tau_2$</td>
<td>$\tau_3$</td>
<td>$\tau_4$</td>
<td>$\tau_5$</td>
</tr>
<tr>
<td>0.015</td>
<td>0.031</td>
<td>0.124</td>
<td>0.062</td>
<td>0.156</td>
</tr>
<tr>
<td>0.2</td>
<td>1.2</td>
<td>1.0</td>
<td>2.0</td>
<td>0.3</td>
</tr>
<tr>
<td>0.48325</td>
<td>0.5637</td>
<td>0.5796</td>
<td>0.3789</td>
<td>0.8820</td>
</tr>
<tr>
<td>0.7</td>
<td>1.16</td>
<td>1.04</td>
<td>1.15</td>
<td>0.59</td>
</tr>
</tbody>
</table>

**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 $F_A$, $F_B$, and $F_C$ are computed by the task composition procedure for each of the three levels of service ($m_{F_A} = m_{F_B} = m_{F_C} = 3$). 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 $d_1$ is the delay at which the capacity has to be 1. The interface $(F_A \parallel F_B) \parallel S_4$ 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 33 = 27, levels of service.

![Figure 10. $(F_A \parallel F_B) \parallel S_5$](image-url)

If the system is composed of the components $a$, $b$, and $C$, as shown in Fig. 12, the resulting interfaces $(F_a \parallel F_b) \parallel S_4$ and $(F_a \parallel F_b) \parallel S_4$ 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 $((F_a \parallel F_b) \parallel F_C) \parallel S_5 = (F_a\parallel(F_b \parallel F_C)) \parallel S_5$, and $(F_a \parallel F_b) \parallel F_C) \parallel S_5 = (F_a \parallel (F_b \parallel F_C)) \parallel S_5$, the interfaces $(F_a \parallel F_b) \parallel F_C) \parallel S_5$ and $(F_a \parallel F_b) \parallel F_C) \parallel S_5$ are not equivalent.
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 $F = (F_a || F_b || F_C) \circ S_5$ 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 $\pi_5$ 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 $\pi_4$ in interface $F'_b$ results in $c_{F'_b}(0) = 0.45$, and 110% of the nominal rate for $\pi_4$ 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, $F' = (F'_a || F'_b || F'_C) \circ S_5$, refines the original specification interface $F$. In particular, the refined interface $F'$ allows for higher rates of the task sequences $\pi_4$ and $\pi_5$ 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
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


