Real-Time Interfaces for Composing Real-Time Systems

Lothar Thiele, Ernesto Wandeler, Nikolay Stoimenov
Computer Engineering and Networks Laboratory, ETH Zurich
8092 Zurich, Switzerland
thiele@tik.ee.ethz.ch, wandeler@tik.ee.ethz.ch, stoimenov@tik.ee.ethz.ch

ABSTRACT

Recently, a number of frameworks were proposed to extend interface theory to the domains of single-processor and distributed real-time systems. This paper unifies some of these approaches and proves properties like refinement and independent implementability. We also explicitly state the requirements to a framework for these properties to be fulfilled. Further, a new notion of adaptive interfaces is introduced that supports the design by providing mechanisms for propagating system constraints, such as (end-to-end) delays, available computing and communication resources, buffer spaces, and energy. Guarantees and assumptions on interfaces are not any longer static but adapt according to the system environment. This can be used to answer synthesis questions at design time or to adapt system parameters to changing environment requirements at run-time. The applicability of the presented framework is proven by adapting it to a number of different real-time analysis models.

Categories and Subject Descriptors: C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems — real-time and embedded systems

General Terms: Design, Performance, Theory

Keywords: Performance Analysis, Real-time Interfaces, Adaptive Interfaces, Interface-based Design

1. INTRODUCTION

One of the major challenges in the design process of complex real-time embedded systems remains to analyze essential characteristics of a system architecture at an early design stage, to support the choice of important design decisions before much time is invested in detailed implementations. Essential characteristics are thereby for example whether maximum delay and throughput constraints are met, what the on-chip memory requirements are, or how different architectural elements must be dimensioned.

The analytical framework for system level performance analysis that was proposed in [12] uses a number of well-known abstractions to capture the timing behavior of event streams and provides additional interfaces between them. Traditional schedulability analysis results are then used to analyze a component-based real-time system design. The framework presented in [19] on the other hand, uses the notion of traditional software component standards such as CORBA and proposes extensions for the support of real-time services, while the approach proposed in [13], composes real-time components by making use of hierarchical scheduling. Finally, the approach to modular performance analysis proposed in [3] relies on Network Calculus [10] and its extension to the domain of real-time embedded systems, Real-Time Calculus [15]. It is based on a general event and resource model, allows to analyze complex systems with hierarchical scheduling and arbitration, and can take computation as well as communication resources into account.

Common to all of these performance analysis methods is that they are applied to analyze a component-based real-time system design a posteriori. That is, while a real-time system gets designed and dimensioned in a first step, it is only after completion of this first step that performance analysis is applied to the system design in a second step. The analysis result will then give an answer to the binary question whether the system design that was developed in the first step will meet all real-time requirements, or not. A designer must then iterate on these two steps until an appropriate system design is found.

Unlike this two-step approach is the idea of interface-based design [6, 7] that proposes a holistic one-step approach to design and analysis of systems. In interface-based design components are described by component interfaces, and in contrast to an abstract component that models what a component does, a component interface models how a component can be used. Through *input assumptions*, a component interface models the expectations that a component has about the other components in the system and the environment, and through *output guarantees*, a component interface tells the other components in the system and the environment what they can expect from this component. The major goal of a good component interface is then to provide enough information to decide whether two or more components can work together properly, where in the case of component interfaces for real-time system performance analysis the term 'properly' refers to questions like: Does the composed system satisfy all requested real-time properties such as delay and throughput constraints?

Additionally, an interface-based real-time system design approach also benefits from the properties of incremental design and independent implementability that are elemen-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'06, October 22–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-542-8/06/0010 ...$5.00.

tary features of interface-based design. The support of *incremental design* ensures that component interfaces can be composed one-by-one into subsystems in any order. And if any of the subsystems cannot be composed successfully, this already forecloses that the complete system also can not be composed successfully, and therefore can not work properly. *Refinement* on the other hand is very similar to subtyping of classes in OO programming, and a component interface can be refined by another component interface that accepts all inputs of the original interface and that produces only a subset of the original outputs. Fulfilling these constraints ensures that components with compatible interfaces can be refined independently and still remain compatible, thus supporting *independent implementability.*

Besides these properties, some recently proposed interfaces [17, 18] also support *dynamic adaptability.* These interfaces not only expose enough information to resolve the composability with other component interfaces, but also they change their assumptions and guarantees following principles of constraints propagation.

Compared to most other interface theory research that focuses on stateful interfaces [5, 8, 2], the work presented in this paper is based on stateless assume / guarantee (A/G) interfaces. While other recent approaches in the area of real-time systems also rely on stateless A/G interfaces [9], the theory presented in this paper not only generalizes the work presented in [9], but it is also able to empower a large class of real-time systems analysis methods with the principles of interface-based design. The presented theory is thereby an analysis, extension and generalization of the basic real-time interfaces that were first presented in [17].

Contributions of this work:

- The paper presents the framework of real-time interfaces that unifies a large set of modular and interface-based approaches to real-time system design known so far, see Section 5.
- Important aspects like independent implementability, refinement and incremental design are discussed and corresponding conditions are derived, see Sections 2 and 3.
- The new notion of adaptive interfaces supports the design by providing mechanisms to propagate constraints, see Section 3.
- The use of real-time interfaces is discussed using a set of different examples, analysis methods and abstractions, see Section 5.

In the following description of real-time interfaces, we will make a distinction between *abstract components* and *their adaptive interfaces.* Both terms are widely used and there are many interpretations available. Before specifying formally the meaning of both terms in the context of real-time interfaces, let us introduce them informally.

Abstract components describe building blocks for a system-wide analysis. They can represent various composable entities, such as tasks, resources, and scheduling disciplines. Instances of inputs and outputs abstract relevant properties of all first class citizens of the analysis method, such as resource capabilities and event streams. In summary, an abstract component provides a mathematical model of a component.

An interface on the other hand should expose enough information about a component as to make it possible to predict if two or more components can work properly together by looking only at their interfaces [7]. *Adaptive interfaces* as used in the context of real-time systems [17, 18] not only allow for such an analysis of a given real-time system, but also support the design by providing mechanisms to propagate constraints, for example (end-to-end) delays, computing and communication resources, buffer spaces, and energy. Guarantees and assumptions are not any longer static but adapt according to the changing system constraints.

The fact that many well known analysis methods can be represented makes the described framework widely applicable. Therefore, the examples used in the paper should be understood only as illustrations of the main principles.

2. ABSTRACT COMPONENTS

As has been described above, abstract components in the context of real-time interfaces represent building blocks of an analysis method. At first, single abstract components will be described.

2.1 Single Abstract Component

DEFINITION 2.1. *An* abstract component (X, Y, T, Ψ) *has input variables X and output variables Y, respectively, a function T and a predicate Ψ over the input variables X. An abstract component* can work properly *iff Ψ holds for some valuation of input variables.*

The transfer function $T(X)$[1] represents the transformation of input values X to output values Y in the analysis, i.e. $Y = T(X)$. The predicate $\Psi(X)$ restricts the scope in which the underlying component can be used. It formalizes the notion that a component can work properly.

EXAMPLE 2.2. *The running example is not directly related to real-time analysis and serves to explain basic concepts only. We suppose that packet streams share a communication unit. The analysis simply adds the data rates of the streams and requires that the accumulated rates do not exceed the available bandwidth. Fig. 1 represents a corresponding abstract component G where x_2 represents the available bandwidth and x_1 the bandwidth used by a single packet stream. Note that the outgoing packet stream y_1 may trigger a computation demand on a computing unit that is attached to the communication unit. y_2 denotes the remaining bandwidth available to other streams. Ψ_G notes that the component works only if the available bandwidth is larger than the requested one. In this example, the input and output variables are simple numbers only. In general, they could*

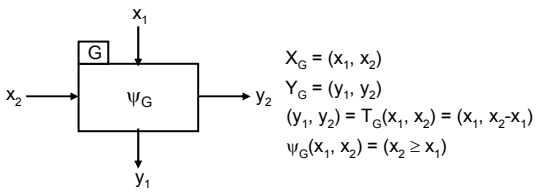

Figure 1: A simple abstract component G representing the analysis of a shared bus.

[1]Unless there are ambiguities, we will not distinguish between a set of variables and their valuation, e.g. depending on the context, X can represent a set of input variables or their values.

however be of any type. In particular, for real-time analysis, we will often use functions over independent variables (arrival and service curves, demand bound functions). Moreover, the predicate may also be used to describe other kinds of constraints such as memory, buffer sizes, energy, and delays.

2.2 Network of Abstract Components

Outputs of abstract components can be connected to inputs which leads to a network of abstract components.[2] The inputs (outputs) of such a network are those inputs (outputs) of its abstract components that are not connected to some output (input).

DEFINITION 2.3. *A* network of abstract components $\mathbb{F}$ *consists of connected abstract components* $F \in \mathbf{F}$ *that form a* connection graph. *If an output is connected to an input, then the variables are identical; un-connected variables are assumed to be different. Connections are point-to-point, i.e. an output is connected to at most one input and vica versa.*[3] *The following quantities are defined:*

- *Inputs of* $\mathbb{F}$: $X_{\mathbb{F}} = (\bigcup_{F\in\mathbf{F}} X_F)\backslash(\bigcup_{F\in\mathbf{F}} Y_F)$
- *Outputs of* $\mathbb{F}$: $Y_{\mathbb{F}} = (\bigcup_{F\in\mathbf{F}} Y_F)\backslash(\bigcup_{F\in\mathbf{F}} X_F)$
- *Predicate of* $\mathbb{F}$: $\Psi_{\mathbb{F}} = \bigwedge_{F\in\mathbf{F}} \Psi_F$
- *The transfer function* $T_{\mathbb{F}}$ *of* $\mathbb{F}$ *is determined by concatenating* T_F, $F \in \mathbf{F}$ *according to the connection graph, i.e.* $Y_{\mathbb{F}} = T_{\mathbb{F}}(X_{\mathbb{F}}) \Rightarrow \bigwedge_{F\in\mathbf{F}}(Y_F = T_F(X_F))$ *is satisfiable for all valuations of* $X_{\mathbb{F}}$.

A network of abstract components can work properly *iff* $\Psi_{\mathbb{F}}$ *is satisfiable for some input* $X_{\mathbb{F}}$.

In this section, we will only look at networks of abstract components whose connection graph does not contain directed cycles. Extensions are described in Section 4. Moreover, it should be noted that a network of abstract components as defined in Section 2.3 is in the form of an abstract component again, i.e. it is defined by sets of input and output variables, a transfer function and a predicate over the input variables.

DEFINITION 2.4. *A* sub-network $\mathbb{G}$ *of a network* $\mathbb{F}$ *is denoted as* $\mathbb{G} \subseteq \mathbb{F}$ *and consists of abstract components in* $\mathbf{G} \subseteq \mathbf{F}$ *that form a subgraph of the connection graph, i.e. deleting abstract components* $\mathbf{F}\backslash\mathbf{G}$ *and incident connections.* Connecting *two networks* $\mathbb{G}$ *and* $\mathbb{H}$ *to form a network* $\mathbb{F}$ *is denoted as* $\mathbb{G} \parallel \mathbb{H} = \mathbb{F}$. *We call* $\mathbb{G}$ *and* $\mathbb{H}$ compatible *(*$\mathbb{G} \sim \mathbb{H}$*) if the network* $\mathbb{G} \parallel \mathbb{H}$ can work properly.

EXAMPLE 2.5. *Fig. 2 represents a simple network of two abstract components of the form defined in Example 2.2. Here, two packet streams with bandwidth* x_1 *and* x_3 *share a common bus with bandwidth* x_2. x_4 *and* y_2 *ought to be just one single variable (to match the previous definitions), but the original names from* G *and* H *are used to simplify the presentation. We have* $X_{\mathbb{F}} = (x_1, x_2, x_3)$, $Y_{\mathbb{F}} = (y_1, y_3, y_4)$, $\Psi_{\mathbb{F}}(x_1, x_2, x_3) = (x_2 \geq x_1) \wedge (x_2 - x_1 \geq x_3)$ *and* $(y_1, y_3, y_4) = T_{\mathbb{F}}(x_1, x_2, x_3) = (x_1, x_3, x_2 - x_1 - x_3)$. *Of course,* $\Psi_{\mathbb{F}}$ *is satisfiable for some input* $X_{\mathbb{F}}$ *and therefore, the abstract components can work properly.*

[2]For the sake of simplicity, the above component model does not define types that would restrict possible compositions. It will be straightforward to extend it.

[3]This is without any restriction in generality. If an output needs to be connected to several inputs, then an additional one-to-many component will do.

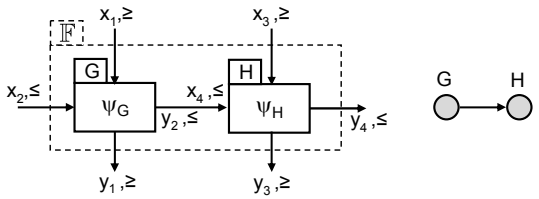

Figure 2: A network of components $\mathbb{F}$ consisting of two abstract components G and H. Associated partial orders and the connection graph are also shown.

2.3 Incremental Design

THEOREM 2.6. *Given a network* $\mathbb{F}$. *If* $\mathbb{F}$ *can work properly then any sub-network* $\mathbb{G} \subseteq \mathbb{F}$ *can work properly.*

PROOF. If $\mathbb{F}$ can work properly, then $\Psi_{\mathbb{F}} = \bigwedge_{F\in\mathbf{F}} \Psi_F$ is satisfied for some input valuation $X_{\mathbb{F}}$, see Def. 2.3. Following the definition of the transfer function $T_{\mathbb{F}}$, we can determine for such an input the values of all internal variables by concatenation of functions T_F, $F \in \mathbf{F}$. For any subnetwork $\mathbb{G} \subseteq \mathbb{F}$, we can set the input variables to the same values as in $\mathbb{F}$. As $\bigwedge_{F\in\mathbf{F}} \Psi_F$ was satisfied, $\bigwedge_{G\in\mathbf{G}\subseteq\mathbf{F}} \Psi_G$ is satisfied too and $\mathbb{G}$ can work properly. □

Following [7], the above theorem ensures the property of *incremental design*, i.e. if a network of abstract components can work properly, then it can be composed in any order from sub-networks and these sub-networks can work properly also. In other words, if a subnetwork can not work properly, then the whole network can not. Obviously, the reverse direction can not be expected: If two networks can work properly, their composition may not.

2.4 Refinement

In a design process, there is often the demand for independent implementability. One can replace the implementation of a sub-network by another one as long as the new abstract component representing this sub-network refines the original one. If the original network could work properly, then this design step should not change this property. This way, the implementation of subsystems can be performed independently.

DEFINITION 2.7. *To each input and output variable of an abstract component, we associate an individual partial order denoted as* $\geq$, *i.e. a binary relation which is reflexive, antisymmetric, and transitive.*[4] *An abstract component* F *is called* monotone, *if for all* $X_F, \tilde{X}_F$

$$X_F \geq \tilde{X}_F \Rightarrow T_F(X_F) \geq T_F(\tilde{X}_F)$$
$$X_F \geq \tilde{X}_F \wedge \Psi_F(X_F) \Rightarrow \Psi_F(\tilde{X}_F)$$

If monotone abstract components are connected, *then the partial orders assigned to connected input/output pairs must be identical.*

It can simply be seen that the composition of monotone abstract components is monotone again as we just compose monotone functions and conjunct monotone predicates. *For*

[4]If we write $X \geq \tilde{X}$ for sets of variables X and $\tilde{X}$, then the comparison is done using the binary relation that is specific for each variable.

the rest of the paper, we will assume that all abstract components are monotone. Based on the notion of monotonicity, we can now define the refinement of an abstract component.

DEFINITION 2.8. *Given a monotone abstract component G that can work properly. Then G'* refines *G ($G \succeq G'$) if*

- *The sets of input and output variables of G and G' are equal.*[5]
- *$(\Psi_G(X) \Rightarrow \Psi_{G'}(X)) \wedge (T_G(X) \geq T_{G'}(X))$ for all valuations of input variables X.*

Note that the abstract component in Def. 2.8 may also describe a whole network, see Def. 2.3. Therefore, the notion of refinement holds for both single abstract components and networks of abstract components.

THEOREM 2.9. *Given a network of monotone abstract components $\mathbb{F}$ that can work properly and an arbitrary partition $\mathbb{F} = \mathbb{G} \parallel \mathbb{H}$. If we refine $\mathbb{G}$ to $\mathbb{G}'$ ($\mathbb{G} \succeq \mathbb{G}'$) then $\mathbb{F}' = \mathbb{G}' \parallel \mathbb{H}$ can work properly, i.e. $\mathbb{G}' \sim \mathbb{H}$.*

PROOF. As $\mathbb{F}$ can work properly, $\bigwedge_{F \in \mathbf{F}} \Psi_F$ can be satisfied for some valuation of input variables $X_{\mathbb{F}}$. The variables in the network with G replaced by G' according to Def. 2.8 are denoted as $\tilde{X}_{G'}$, $\tilde{Y}_{G'}$, $\tilde{X}_H$, $\tilde{Y}_H$. In the original network they are denoted as X_G, Y_G, X_H, Y_H. As all transfer functions are monotone and because of Def. 2.8, we have $\tilde{X}_{G'} \leq X_G$, $\tilde{Y}_{G'} \leq Y_G$, $\tilde{X}_H \leq X_H$, $\tilde{Y}_H \leq Y_H$. Because $(\Psi_G(X_G) \Rightarrow \Psi_{G'}(X_G))$, $\tilde{X}_{G'} \leq X_G$ and the monotonicity of Ψ (see Def. 2.7), the predicates of all abstract components in the new system are satisfied. □

The concepts of refinement and independent implementability will be shown again in the running example.

EXAMPLE 2.10. *Given the small network $\mathbb{F}$ in Fig. 2, we now add partial orders to the different variables. As all variables are simple real numbers and not complex data types such as tuples (e.g. representations of event streams in the form of 'period and jitter' or 'burst size and average rate') or curves (e.g. arrival and service curves), we only use the conventional 'greater or equal' $\geq$ for variables x_1, x_3, y_1, y_3 and 'less or equal' $\leq$ for variables x_2, x_4, y_2, y_4. The partial orders of connected inputs and outputs obviously match. The transfer function $T_{\mathbb{F}}$ and predicate $\Psi_{\mathbb{F}}$ are monotone, see Def. 2.7. For example, as $\Psi_{\mathbb{F}}(x_1, x_2, x_3) = (x_2 \geq x_1) \wedge (x_2 - x_1 \geq x_3)$ we have $x_1 \geq \tilde{x}_1 \wedge x_2 \leq \tilde{x}_2 \wedge x_3 \geq \tilde{x}_3 \wedge [(x_2 \geq x_1) \wedge (x_2 - x_1 \geq x_3)] \Rightarrow (\tilde{x}_2 \geq \tilde{x}_1) \wedge (\tilde{x}_2 - \tilde{x}_1 \geq \tilde{x}_3)$, see also Def. 2.7.*

A refined component G', i.e. $G \succeq G'$, could be characterized by $(y_1, y_2) = T_{G'}(x_1, x_2) = (x_1 - 1, x_2 - x_1 + 1)$ and $\Psi_{G'} = (x_2 + 1 \geq x_1)$. Looking at $T_{G'}$, one can see that the component delivers a packet stream with a smaller bandwidth ($y_1 < x_1$) and provides more communication bandwidth to other packet streams via y_2 ($y_2 > x_2 - x_1 + 1$). If we would replace G in Fig. 2 by G', we would obtain a new network called $\mathbb{F}'$ according to Theorem 2.9. We can now compute the new predicate $\Psi_{\mathbb{F}'}(x_1, x_2, x_3) = (x_2 + 1 \geq x_1) \wedge (x_2 - x_1 + 1 \geq x_3)$. It can easily be verified, that Theorem 2.9 holds. In other words, if predicate $\Psi_{\mathbb{F}}$ holds for a certain valuation of inputs then also $\Psi_{\mathbb{F}'}$ holds and therefore, $\mathbb{F}'$ can work properly.

[5]In order to reduce the notational overhead, we restrict the refinement of an abstract component to changes of its predicate and transfer function only. In a similar way to [7], the number of inputs and outputs could also be changed.

3. ADAPTIVE REAL-TIME INTERFACES

An adaptive interface of an abstract component not only exposes enough information to predict whether it can be composed and work properly. In addition, (a) it is adaptive as it changes assumptions and guarantees depending on the system environment and constraints and (b) it distributes constraints globally through the whole network. At first, we will describe the main concept informally.

The following Fig. 3 shows an abstract component and the corresponding adaptive interface representation. The

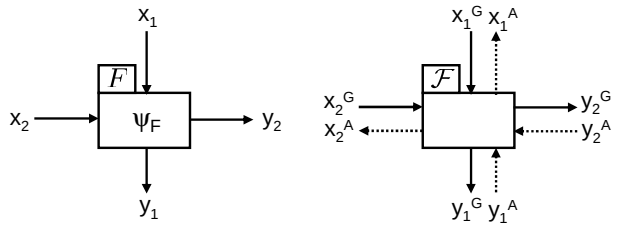

Figure 3: A simple abstract component F and its adaptive interface representation.

abstract component variables represent some abstraction of the actual component behavior, where $X_F = (x_1, x_2)$, $Y_F = (y_1, y_2)$ represent abstractions of its inputs and outputs, respectively. The component works properly if $\Psi_F(X_F)$ is satisfied and it can work properly, if $\Psi_F(X_F)$ is satisfiable.

If we make the transition from an abstract component to its real-time adaptive interface (see Fig. 3), then we still have input and output variables, now called guaranteed values $X_F^G \geq X_F$, $Y_F^G \geq Y_F$. In other words, the network of abstract components works properly whenever the variables are smaller than the guaranteed values in the corresponding network of interfaces. The predicate Ψ_F has been converted into the new assume variables X_F^A: Whenever we have $X_F^A \geq X_F^G$, then Ψ_F is satisfied[6]. The adaptive interface makes the predicate Ψ_F explicit in form of additional *input assume* variables. They appear now at outputs of other adaptive interfaces as *output assume* variables. Therefore, the following interpretation can be given:

- *Input assume variables*: If we have $X_F^A \geq X_F^G$, then (a) the corresponding component works properly and (b) the requests of all connected components (indicated by Y_F^A) are satisfied, i.e. $Y_F^A \geq Y_F^G$.
- *Output assume variables*: The output assume variables Y_F^A describe the assumption of the environment (e.g. other components) towards F, i.e. they request that $Y_F^A \geq Y_F^G$.

Therefore, depending on system inputs, constraints and requirements, a particular combination of assumptions and guarantees is determined for each component. This way, it is possible to represent adaptive behavior of a component, e.g. if it changes its behavior depending on *local* requirements (captured by the input guarantees and output assumes). Moreover, one can decide whether the whole system is able to meet constraints and under what assumptions on its inputs, thereby solving important synthesis questions. Examples are given in Example 3.14 and Section 5.

[6]One may extend the framework and consider predicates in the abstract interface also, see Section 4.

3.1 Single Adaptive Interface

DEFINITION 3.1. *An* adaptive interface $(X^A, X^G, Y^A, Y^G, T^f, T^b)$ *corresponds to a monotone abstract component* (X, Y, T, Ψ) *with at least one input and is characterized as follows:*

- *A set of input guarantee and input assume variables* X^G *and* X^A*, one for each variable in the inputs* X *of the abstract component.*
- *A set of output guarantee and output assume variables* Y^G *and* Y^A*, one for each variable in the outputs* Y *of the abstract component.*
- *A monotone forward transfer function with* $Y^G = T^f(X^G)$.
- *A backward transfer function with* $X^A = T^b(X^G, Y^A)$.

In addition, we require that

$$X^A \geq X^G \geq X \Rightarrow Y^A \geq Y^G \geq Y \wedge \Psi(X)$$

If the input values of the abstract component as well as the corresponding assumptions and guarantees match (using the partial order assigned to each single input), then the same holds for the output values and the predicate of the abstract component is satisfied. Later on, we will determine T^f and T^b such that this requirement is satisfied. But at first, the connection of two interfaces $\mathcal{G}$ and $\mathcal{H}$ will be defined.

EXAMPLE 3.2. *Fig. 4 shows on the left hand side an adaptive interface that corresponds to the abstract component shown in Fig. 1. We have* $X^G_{\mathcal{G}} = (x^G_1, x^G_2)$,

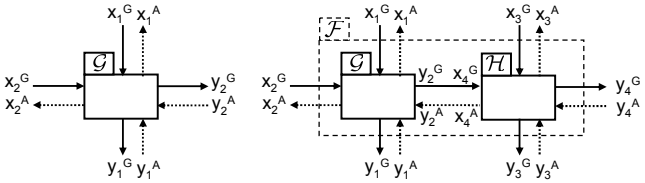

Figure 4: Adaptive interfaces that corresponds to the abstract component and network of abstract components shown in Fig. 1 and Fig. 2, respectively.

$X^A_{\mathcal{G}} = (x^A_1, x^A_2)$, $Y^G_{\mathcal{G}} = (y^G_1, y^G_2)$ *and* $Y^A_{\mathcal{G}} = (y^A_1, y^A_2)$. *Let us use the forward and backward transfer functions* $(y^G_1, y^G_2) = T^f_{\mathcal{G}}(x^G_1, x^G_2) = (x^G_1, x^G_2 - x^G_1)$ *and* $(x^A_1, x^A_2) = T^b_{\mathcal{G}}(x^G_1, x^G_2, y^A_1, y^A_2) = (\min\{x^G_2, y^A_1\}, \max\{x^G_1, y^A_2 + x^G_1\})$. *Then we can easily check that* $X^A_{\mathcal{G}} \geq X^G_{\mathcal{G}} \geq X_{\mathcal{G}} \Rightarrow Y^A_{\mathcal{G}} \geq Y^G_{\mathcal{G}} \geq Y_{\mathcal{G}} \wedge \Psi_{\mathcal{G}}(X_{\mathcal{G}})$, *i.e. if the assumptions and guarantees of the inputs match* ($x^A_1 \geq x^G_1$,$x^A_2 \leq x^G_2$), *then those of the outputs do the same* ($y^A_1 \geq y^G_1$,$y^A_2 \leq y^G_2$) *and the predicate of the abstract component is satisfied* ($\Psi_{\mathcal{G}}(X^G_{\mathcal{G}}) = (x^G_2 \geq x^G_1)$).

3.2 Connecting Adaptive Interfaces

DEFINITION 3.3. *The* connection *of two interfaces* $\mathcal{G} \parallel \mathcal{H}$ *follows the connection of the corresponding monotone abstract components, i.e.* $G \parallel H$. *The forward and backward transfer functions* $T^f_{\mathcal{F}}$ *and* $T^b_{\mathcal{F}}$ *are determined by composing* $T^f_{\mathcal{G}}$ *with* $T^f_{\mathcal{H}}$, *and* $T^b_{\mathcal{G}}$ *with* $T^b_{\mathcal{H}}$, *respectively, following the connections of the interface inputs and outputs, see Def. 2.3.*

The above definition can easily be extended to the case of a network of adaptive interfaces as in the case of abstract components, see Def. 2.3. The construction of the forward and backward transfer functions T^f and T^b according to Def. 3.3 by a simple concatenation of functions requires that there are no dependency cycles.

THEOREM 3.4. *Given a network of monotone abstract components* $\mathbb{F}$ *whose connection graph is free of directed cycles and a partitioning* $\mathbb{F} = \mathbb{G} \parallel \mathbb{H}$. *Interfaces* $\mathcal{G}$ *and* $\mathcal{H}$ *correspond to the subnetworks* $\mathbb{G}$ *and* $\mathbb{H}$, *respectively. Then* $T^f_{\mathcal{F}}$ *and* $T^b_{\mathcal{F}}$ *can be determined without dependency cycles, i.e. by simple concatenation.*

PROOF. The theorem can be derived from properties of the connection graph. □

From Def. 3.3, we know how to combine a set of adaptive interfaces to a new adaptive interface. In addition, Def. 3.1 states the condition, that an adaptive interface actually represents an abstract component, i.e. that it can work properly. The above concepts are useful only, if we are able to show that the combined adaptive interface contains enough information to decide whether all abstract components it represents can work properly together.

THEOREM 3.5. *The composition of two interfaces* $\mathcal{F} = \mathcal{G} \parallel \mathcal{H}$ *is called compatible* ($\mathcal{G} \sim \mathcal{H}$), *iff* $X^A_{\mathcal{F}} \geq X^G_{\mathcal{F}}$ *is satisfiable. Then we have* $(\mathcal{G} \sim \mathcal{H}) \Rightarrow (\mathbb{G} \sim \mathbb{H})$, *i.e. the corresponding abstract components also can work properly together. In particular, we have*

$$X^A_{\mathcal{F}} \geq X^G_{\mathcal{F}} \geq X_{\mathbb{F}} \Rightarrow Y^A_{\mathcal{F}} \geq Y^G_{\mathcal{F}} \geq Y_{\mathbb{F}} \wedge \Psi_{\mathbb{F}}(X_{\mathbb{F}})$$

PROOF. Let us first consider two connected abstract components M and N, i.e. $\mathbb{K} = M \parallel N$. As the connection graph does not contain any directed cycles, we can suppose that only outputs of M are connected to inputs of N. The corresponding adaptive interfaces are connected correspondingly. For $\mathcal{M}$ we have $X^A_{\mathcal{M}} \geq X^G_{\mathcal{M}} \geq X_M \Rightarrow Y^A_{\mathcal{M}} \geq Y^G_{\mathcal{M}} \geq Y_M \wedge \Psi_M(X_M)$. As some of the outputs of M are connected to some inputs of N and the other inputs of N are inputs of $\mathbb{K}$, we also have $X^A_{\mathcal{K}} \geq X^G_{\mathcal{K}} \geq X_{\mathbb{K}} \Rightarrow X^A_{\mathcal{N}} \geq X^G_{\mathcal{N}} \geq X_N \wedge \Psi_M(X_M)$. If we apply similar arguments to N, we obtain $X^A_{\mathcal{K}} \geq X^G_{\mathcal{K}} \geq X_{\mathbb{K}} \Rightarrow Y^A_{\mathcal{K}} \geq Y^G_{\mathcal{K}} \geq Y_{\mathbb{K}} \wedge \Psi_{\mathbb{K}}(X_{\mathbb{K}})$. The same argument can be now recursively applied to the case of interfaces that represent sub-networks of abstract components. □

EXAMPLE 3.6. *We are continuing Example 3.2 by connecting the adaptive interfaces* $\mathcal{G}$, $\mathcal{H}$ *of the abstract components G, H according to the network* $\mathbb{F}$ *shown in Fig. 2. The resulting interface* $\mathcal{F}$ *and its composition* $\mathcal{F} = \mathcal{G} \parallel \mathcal{H}$ *are shown in Fig. 4 on the right hand side. As stated in Theorem 3.4, the resulting forward and backward transfer functions can be computed by simple composition. We obtain* $(y^G_1, y^G_3, y^G_4) = T^f(X^G_{\mathcal{F}}) = (x^G_1, x^G_3, x^G_2 - x^G_1 - x^G_3)$ *and* $(x^A_1, x^A_2, x^A_3) = T^b(X^G_{\mathcal{F}}, Y^A_{\mathcal{F}}) = (\min\{x^G_2, y^A_1\}, \max\{x^G_1, x^G_1 + x^G_3, x^G_1 + x^G_3 + y^A_4\}, \min\{x^G_2 - x^G_1, y^A_3\})$. *One can easily check that* $X^A_{\mathcal{F}} \geq X^G_{\mathcal{F}}$ *is satisfiable and the abstract components can work together properly.*

3.3 Transfer Functions

So far, we did not determine the forward and backward functions T^f and T^b such that the relation $X^A \geq X^G \geq X \Rightarrow Y^A \geq Y^G \geq Y \wedge \Psi(X)$ in Def. 3.1 holds.

THEOREM 3.7. *If for all X, Y we have $T^f(X) \geq T(X)$ and $X \leq T^b(X,Y) \Rightarrow Y \geq T(X) \wedge \Psi(X)$ then $X^A \geq X^G \geq X \Rightarrow Y^A \geq Y^G \geq Y \wedge \Psi(X)$.*

PROOF. We have $X^A \geq X^G \geq X \Rightarrow T^b(X^G, Y^A) \geq X^G \wedge T^f(X^G) \geq T^f(X) \Rightarrow Y^A \geq T(X^G) \wedge Y^G \geq Y \wedge \Psi(X^G) \Rightarrow Y^A \geq Y^G \geq Y \wedge \Psi(X)$. Here we make use of the monotonicity of T and Ψ. □

The above theorem leads to a simple constructive method to determine T^f, namely $T^f = T$. In case of T^b, we will determine one possibility next. The input and output variables of the abstract component are denoted as $X = (x_1, ..., x_N)$. Then the construction of a large but feasible $T^b(X,Y)$ involves three steps:

1. Determine a set of N functions $\widetilde{\Psi}_i(X)$ such that $\widetilde{\Psi}_i(X) = \max\{z \mid \Psi(x_1, ..., x_i, z, x_{i+1}, ..., x_N)\}$ for all X.
2. Determine a set of N functions $\widetilde{T}_i(X,Y)$ such that $\widetilde{T}_i(X,Y) = \max\{z \mid Y \geq T^f(x_1, ..., x_i, z, x_{i+1}, ..., x_N)\}$ for all X, Y.
3. Finally, we have for all i: $T_i^b(X,Y) = \inf\{\widetilde{\Psi}_i(X), \widetilde{T}_i(X, Y)\}$ for all X, Y.

In step 1 and 2, max denotes a maximal element of a set where the partial order relation of x_i is used. Let us choose some X, Y such that $X \leq T^b(X,Y)$ holds, then Step 3 yields $x_i \leq \widetilde{\Psi}_i(X)$ and $x_i \leq \widetilde{T}_i(X,Y)$. Because of 1 we find that $\Psi(X)$ holds because $\Psi(x_1, ..., x_{i-1}, \tilde{\Psi}_i(X), x_{i+1}, ..., x_N)$ is satisfied for all $1 \leq i \leq N$, $x_i \leq \widetilde{\Psi}_i(X)$ and Ψ is monotone. Because of 2 we find that $Y \geq T(X)$ for all i holds because $Y \geq T(x_1, ..., x_{i-1}, \widetilde{T}_i(X,Y), x_{i+1}, ..., x_N)$, $x_i \leq \widetilde{T}_i(X,Y)$ and T is monotone.

EXAMPLE 3.8. *The forward and backward transfer functions for $\mathcal{G}$ as used in Example 3.2 have been determined by the above method. Obviously, we had $T^f_{\mathcal{G}} = T_G$, see also Fig. 1. Let us now construct the $x_2^A = T_2^b(X_{\mathcal{G}}^G, Y_{\mathcal{G}}^A)$. For step 2 we obtain $\tilde{T}_{21} = \min\{z \mid y_1^A \geq x_1^G\} = -\infty$ and $\tilde{T}_{22} = \min\{z \mid y_2^A \leq z - x_1^G\} = y_2^A + x_1^G$. For step 1 we obtain $\tilde{\Psi}_2 = \min\{z \mid z \geq x_1^G\} = x_1^G$. Note that we have to consider the correct partial orders everywhere. Finally, we obtain $x_2^A = T_2^b(X_{\mathcal{G}}^G, Y_{\mathcal{G}}^A) = \sup\{\tilde{\Psi}_2, \tilde{T}_{21}, \tilde{T}_{22}\} = \max\{x_1^G, y_2^A + x_1^G\}$. This is the same expression as used in Example 3.2.*

3.4 Refinement

In a similar way to the refinement of abstract components, we can define the refinement of adaptive interfaces. Let us suppose that a component is implemented together with an implementation of its adaptive interface. This way, at a system house (that combines the independently implemented components) one can connect the interfaces during design time in order to check whether the components work properly for the specific environment and set of constraints. One may even adapt the assumptions and guarantees at run-time in order to perform a system-wide admittance test in case the environment changes (adaptive behavior). In this case, the interfaces are actually implemented on the run-time system. In both cases it would be useful if the implementation of such a combined component/interface can be performed independently, i.e. involving also a possible change in the interface.

The following arguments follow closely those in Section 2.4.

DEFINITION 3.9. *Given an adaptive interface $\mathcal{G}$. Then $\mathcal{G}'$* refines *$\mathcal{G}$ ($\mathcal{G} \succeq \mathcal{G}'$) if*

- *The sets of input and output variables of $\mathcal{G}$ and $\mathcal{G}'$ are equal.*
- *$T^f_{\mathcal{G}}(X)$ is monotone in X and $T^b_{\mathcal{G}}(X,Y)$ is monotone in Y, antitone in X.*
- *$T^f_{\mathcal{G}}(X) \geq T^f_{\mathcal{G}'}(X)$ and $T^b_{\mathcal{G}}(X,Y) \leq T^b_{\mathcal{G}'}(X,Y)$ for all valuations of variables X, Y.*

From Def. 3.9 it follows that a refined adaptive interface has a weaker forward and a stronger backward transfer function. Now we can state the main refinement theorem.

THEOREM 3.10. *Given adaptive interfaces and a compatible connection, i.e. $\mathcal{F} = \mathcal{G} \parallel \mathcal{H}$ and $\mathcal{G} \sim \mathcal{H}$. If we refine $\mathcal{G}$ to $\mathcal{G}'$ ($\mathcal{G} \succeq \mathcal{G}'$) then $\mathcal{F}' = \mathcal{G}' \parallel \mathcal{H}$ is a compatible connection, i.e. $\mathcal{G}' \sim \mathcal{H}$.*

PROOF. The proof follows the principles used in that of Theorem 3.5. □

3.5 Incremental Design

We would ideally expect, that a similar (positive) result, as for networks of abstract components, holds in the case of adaptive interfaces. But the property of incremental design does not hold for adaptive interfaces as defined so far. Composition of adaptive interfaces is not associative and therefore, the success of a design (in terms of compatibility of interfaces) depends on the ordering of composition.

On the other hand, if we would have a one-to-one correspondence between the compatibility of adaptive components and interfaces, then the property stated in Theorem 2.6 would carry over to adaptive interfaces.

DEFINITION 3.11. *A* strict adaptive interface *is an adaptive interface according to Def. 3.1 that satisfies in addition*

$$Y^A \geq Y^G \wedge \Psi(X^G) \Rightarrow X^A \geq X^G$$

Now, we can state the necessary strong relation between abstract components and strict adaptive interfaces.

THEOREM 3.12. *Given is a composition of two strict interfaces $\mathcal{F} = \mathcal{G} \parallel \mathcal{H}$ where the corresponding network of components $\mathbb{F}$ does not have any output[7]. Then we have $(\mathcal{G} \sim \mathcal{H}) \Leftrightarrow (\mathbb{G} \sim \mathbb{H})$.*

PROOF. The proof follows the arguments given in the proof of Theorem 3.5. □

As a result of Theorem 3.12, there is a one-to-one correspondence between the compatibility relations between interfaces and the corresponding underlying network of abstract components. Therefore, if adaptive interfaces are strict, then they allow for incremental design as defined in [7].

It remains to show, how we can construct the transfer functions of a strict adaptive interface, and when they exist.

[7]This technical condition ensures that we do not impose additional assume/guarantee constraints from the environment that are not present in the corresponding network of abstract components. This is no restriction of generality as one could close open outputs with additional abstract components.

THEOREM 3.13. *We call T^f and T^b strict if they satisfy the conditions of Theorem 3.7 and in addition $\Psi(X) \Rightarrow T^b(X, T(X)) \geq X$ for all X. In this case we have $X^A \geq X^G \Leftrightarrow Y^A \geq Y^G \wedge \Psi(X^G)$.*

PROOF. The forward direction has already been proven in Theorem 3.7. Here we show: $Y^A \geq Y^G \wedge \Psi(X^G) \Rightarrow X^A \geq T^b(X^G, Y^G) \wedge \Psi(X^G) \Rightarrow X^A \geq T^b(X^G, T(X^G)) \wedge \Psi(X^G) \Rightarrow X^A \geq T^b(X^G, T(X^G)) \geq X^G$. □

Finally, we will give a constructive method to determine strict forward and backward transfer functions from T and Ψ. It follows directly the approach taken in Section 3.3 but replaces 'a maximal' element by 'the greatest' element:

$$T^f(X) = T(X)$$
$$\widetilde{\Psi}_i(X) = grt\{z \mid \Psi(x_1, ..., x_{i-1}, z, x_{i+1}, ..., x_N)\}$$
$$\widetilde{T}_i(X, Y) = grt\{z \mid Y \geq T(x_1, ..., x_{i-1}, z, x_{i+1}, ..., x_N)\}$$
$$T_i^b(X, Y) = \inf\{\widetilde{\Psi}_i(X), \widetilde{T}_i(X, Y)\}$$

where grt yields the greatest element of a set (if it exists) and inf denotes the infimum of a set. Note that the sets are partially ordered using the partial order associated to each input and output, see Def. 2.7. For example, inf and grt in the above equations use the partial order associated with x_i. It also appears, that T^b can only be computed if the greatest elements of the partially ordered sets exist.

One direction of a proof of correctness is given in Section 3.3 already. Here, we show the following implications for some X: $T^b(X, T(X)) = \inf\{\widetilde{\Psi}(X), \widetilde{T}(X, T(X))\} \not\geq X \Rightarrow \widetilde{T}(X, T(X)) \not\geq X \vee \widetilde{\Psi}(X) \not\geq X \Rightarrow \text{false} \vee \overline{\Psi(X)}$.

EXAMPLE 3.14. *As the forward and backward transfer functions of $\mathcal{G}$ have been already determined in the way described here, they are strict and therefore, the adaptive interface shown in Fig. 4 and Example 3.2 is strict too. As a result from Theorem 3.12 we can also conclude that the interface $\mathcal{F}$ as shown on the right hand side of Fig. 4 is strict and we have the compatibility relation $(\mathcal{G} \sim \mathcal{H}) \Leftrightarrow (G \sim H)$, see Theorem 3.12. The corresponding transfer functions are given in Example 3.6.*

Finally, we show how one can use adaptive interfaces for solving design problems off-line or for adapting to changes in the environment that are either caused by changed requirements (assumptions) or changed properties. Fig. 5 shows the adaptive interface $\mathcal{F}$ again, but now with some (assumed) valuations of outputs. Now, we can summarize some uses

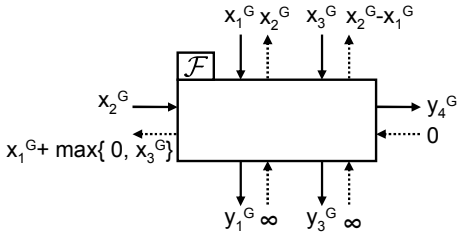

Figure 5: Adaptive interface as in Fig. 4, but with valuations for output assumptions and computed values for input assumptions.

of this adaptive interfaces at design time and at run-time (if implemented on the target system).

Design Time*: Let us suppose, that there are two packet streams with bandwidth requirements x_1^G and x_3^G to be processed, then we can directly compute the requirement towards the bus bandwidth x_2^A. As in the above example, x_2^A does not depend on x_2^G, see Section 4, we can also set the bus bandwidth to its minimal feasible value $x_2^G = x_1^G + \max\{0, x_3^G\}$, given input demands x_1^G and x_3^G of the traffic streams. After any single change of one of the input guarantees or output assumptions, we need to recalculate all the other assumptions and guarantees.*

Run-Time*: If the adaptive interface is implemented in the run-time system, the above described process can be used to adapt to new requirements and guarantees of the environment:*

1. *Change* one *input guarantee according to a new environment, e.g. adapt the available bus bandwidth or the bandwidth of packet streams while respecting the corresponding assume.*
2. *Recalculate the resulting assumptions in the whole system in order to adapt to the new environment.*

This method can be applied, if the underlying network of abstract components is free of undirected cycles, see Section 4.

Finally, Fig. 6 shows a more complex scenario that shows the modularity of the framework described in the paper. Here, we have two independent resources such as a commu-

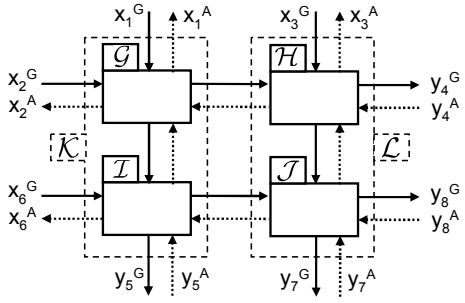

Figure 6: Adaptive interface for a more complex system with two resources.

nication unit with bandwidth x_2 and a computing resource with service x_6. Both packet streams pass the communication unit and the computing device. Note that the interfaces could be combined in any order (incremental design property) because of the strict forward and backward transfer functions. As indicated in Fig. 6, we could construct interfaces $\mathcal{K}$ and $\mathcal{L}$ that abstract each packet stream (instead of abstracting the resources as in Fig. 5).

4. EXTENSIONS

The above framework for interface-based analysis, design and adaptation of real-time systems can be extended in several ways. Because of lack of space, the following discussion is done in an informal manner and only the major ideas and concepts are presented.

Multiple Assume-Guarantee Pairs. In the adaptive interfaces defined so far, there has been exactly one assume-guarantee pair for each input and output of the corresponding abstract component. One could extend the framework by allowing several of this pairs. This way, it is possible to

consider for example upper and lower constraints for a given value. This has been extensively used in the interface-based design of hierarchical scheduling, see [18].

Additional Predicates in Adaptive Interfaces. Clearly, not all constraints imposed by an environment or by abstract components can be simply phrased into the restricted class of adaptive interfaces we defined. It is possible to add to each interface additional predicates (like the ones defined for abstract components).

Cycles in Networks of Abstract Components. We have been restricting the paper to the case where there are no directed cycles in the network of components. If we even have *no undirected cycles*, then the network of adaptive interfaces has a very interesting and practically important property: For any input, the assume value does not change if the corresponding guarantee is changed and for any output, the guarantee value does not change if the corresponding assume is changed. In such a network (no undirected cycles) of interfaces where forward and backward transfer functions are calculated according to Section 3.5, we can select an input guarantee x^G and change it while respecting $x^G \leq x^A$. This can be used during design time in order to determine e.g. the maximally allowed input rates or the minimal processor capabilities. And when used on-line, the change of the environment at a single input can be accepted as long as $x^G \leq x^A$. After re-calculation of the interface equations, new assumptions towards the environment are calculated. The same principle does hold for output assumptions too.

If we consider directed cycles, the computation of the transfer function of connected components may require fixed point calculations. The same holds for the forward and backward transfer functions of the corresponding interfaces. The present framework provides the necessary algebraic background to formally argue about the existence of these fixed points. Note that fixed point calculations in the context of real-time analysis have been in use for some time [12].

5. APPLICATIONS

5.1 A Comprehensive EDF Example

In this subsection, we will describe the application of the real-time interface framework to a single processor real-time scheduling scenario using earliest deadline first principles. It is supposed that the reader is familiar with the basic analysis methods based on demand and resource bound functions as described for example in [1] and using bounded delay models, e.g. [11]. The scenario we are looking at is identical to that described in [9]:

- Given a single processor characterized by a service curve $\beta(\Delta)$ (also denoted as supply bound function) which denotes the minimal computation time available in any time interval of length Δ. For example, a processor is characterized by $\beta(\Delta) = \Delta$.
- A set of tasks τ_i with computation time e_i and deadline d_i.
- The tasks are activated using event streams S_j that are described using arrival curves $\alpha_j(\Delta)$ (also denoted as resource bound function). Here, $\alpha_j(\Delta)$ denotes the maximal number of events in any time interval of length Δ. For example, a periodic event stream with period p_j has $\alpha_j(\Delta) = \lceil \frac{\Delta}{p_j} \rceil$.
- An event stream triggers a chain of tasks, i.e each event in a stream directly triggers the first task. When an event has been processed by a task, it triggers the next task in the chain.

As an example, we will first describe the use of abstract components and adaptive interfaces in this scenario, based on the modular performance analysis method described in [3],[17] and [18]. Let us suppose that a task t_i is activated by the events of several event streams, each one characterized by an arrival curve $\alpha_k(\Delta)$, $k \in \mathbf{K}_i$. Then it is activated by the accumulated arrival curve $\alpha_i(\Delta) = \sum_{k \in \mathbf{K}_i} \alpha_k(\Delta)$ which is the sum of the arrival curves of all activating streams. A set of tasks τ_i, $i \in \mathbf{I}$ is schedulable by EDF, iff $\beta(\Delta) \geq \sum_{i \in \mathbf{I}} e_i \cdot \alpha_i(\Delta - d_i)$ for all $\Delta > 0$[8]. The output stream of a task τ_i that contains processed events from the stream with arrival curve $\alpha_k(\Delta)$, $k \in \mathbf{K}_i$ is bounded by the arrival curve $\alpha'_k(\Delta) = \alpha_k(\Delta + (d_i - e_i))$ for any $k \in \mathbf{K}_i$.

Using these well known facts, we can now build networks of abstract components that correspond to any particular scenario. To this end, we define two basic abstract components as shown in Fig. 7 on the left hand side.

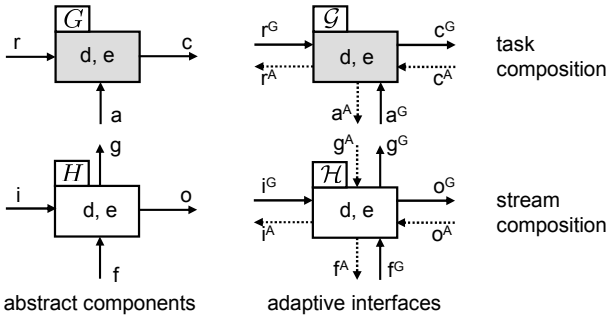

Figure 7: Basic abstract components and interfaces for EDF scheduling.

The top component describes the use of the available service r to process a task characterized by deadline d and computation time e. Therefore, variable r carries a service curves $\beta(\Delta)$, a carries the accumulated arrival curves of the task $\alpha(\Delta)$. According to the above analysis, the output c carries $\beta(\Delta) - e \cdot \alpha(\Delta - d)$ which now is the transfer function of the abstract task component. The predicate of the abstract task component is $\beta(\Delta) \geq e \cdot \alpha(\Delta - d)$. Note that the above transfer function and predicate are monotone with respect to the partial order $\leq$ for r, c where $a \leq b$ iff $a(\Delta) \leq b(\Delta)$ for all Δ and $\geq$ for a (defined in a similar way). Defining the operator $\triangleright$ as

$$(a \triangleright d)(\Delta) = \begin{cases} a(\Delta - d) & \Delta > \max(d, 0) \\ 0 & 0 \leq \Delta \leq \max(d, 0) \end{cases}$$

we can also write $c = r - e \cdot (a \triangleright d)$ and $\Psi = (r \geq e \cdot (a \triangleright d))$.

The bottom abstract component denotes the processing of a stream with arrival curve $\alpha(\Delta)$ which is associated to input i. Using the above shorthand notation, we find $o = i \triangleright (e - d)$ (which is the output arrival curve $\alpha(\Delta + (d - e))$) and $g = i + f$ (which is the accumulation of arrival curves of activating streams, i.e. from the new input i and from other inputs

[8]We suppose that $\alpha(\Delta) = 0$ for all $\Delta \leq 0$. The right hand side is also denoted as demand bound function.

f). Besides these (monotone) transfer functions, there is no predicate associated to a stream component.

By combining the above two classes of components according to a given scenario, we get a network of abstract components that can be used as described in the previous section. An example is given in Fig. 8 on the left hand side.

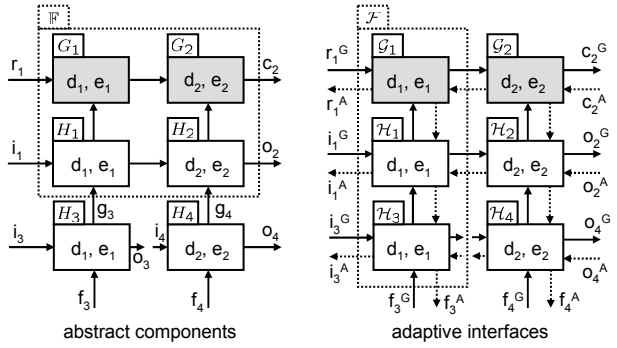

Figure 8: Example of a network of abstract components and interfaces for EDF scheduling.

The scenario consists of two tasks τ_1 and τ_2 (there are two task components). There are 3 event streams, namely i_1, i_3 and i_4 where i_1 at first passes τ_1 and then τ_2. τ_1 is triggered by i_1 and i_3, and τ_2 is triggered by i_1 (after passing τ_1) and by i_4. Because of the independent implementability property, see Theorem 2.6, any partitioning in sub-networks is possible (commutativity and associativity of composition). For example, $\mathbb{F}$ denotes an abstract component that describes the resource usage by the two tasks including the processing of stream i_1. The corresponding transfer function and predicate is obtained by a (trivial) composition of the individual transfer functions of the used abstract components G_1, G_2, H_1 and H_2.

In order to allow for solving synthesis problems and propagating constraints, we can also construct the adaptive interfaces corresponding to the abstract components, see Fig. 7 on the right hand side. According to Theorem 3.13 and the subsequent method, we can construct strict forward and backward transfer functions. To this end, we need to define the 'inverse' operator to $\triangleright$, i.e.

$$(a \triangleleft d)(\Delta) = \begin{cases} a(\Delta - d) & \Delta > d \\ a(0^+) & 0 < \Delta \leq d \\ 0 & \Delta = 0 \end{cases}$$

Then we find for the adaptive task interface $c^G = r^G - e \cdot (a^G \triangleright d)$, $a^A = ((r^G - c^A) \triangleright (-d))/e$ and $r^A = c^A + e \cdot (a^G \triangleright d)$. For the adaptive stream interface we find $g^G = i^G + f^G$, $o^G = i^G \triangleright (e - d)$, $f^A = g^A - i^G$ and finally, $i^A = \min\{g^A - f^G, o^A \triangleleft (d - e)\}$. The above forward and backward transfer functions are strict and therefore, the composition of the adaptive interfaces is associative and commutative (independent implementability). Note that the method described in [9] does not guarantee this property. Fig. 8 presents the same scenario, but now with a different hierarchical component, namely $\mathcal{F}$ which abstracts all activations of task τ_1 and its influence on the system behavior. Note that the combined interface can simply be constructed by composing the forward and backward transfer functions. The system environment can be closed by setting $f_3^G = f_4^G = 0$, $o_2^A = o_3^A = o_4^A = \infty$ and $c_2^A = 0$ for example. Because of the adaptivity of the representation, the suitability of any other environment with other guarantees and assumptions can simply be checked.

Finally, one should note, that the same example could also be handled with a different abstraction of resources and event streams, based on the well known *bounded delay model*, see e.g. [11]. In this case, the structure as shown in Figs. 7 and 8 remains the same, only the data types associated to the variables (i.e. the abstraction used in the analysis) as well as the partial orders, transfer functions and predicates change. The bounded delay model is a much coarser abstraction and therefore, the bounds obtained are worse. On the other hand, the analysis and synthesis is computationally much simpler. The transfer functions and predicates can now be determined simply by combining the results described in this paper and [9].

5.2 Hierarchical Scheduling

In [14], Shin et al. propose a compositional scheduling framework to determine the schedulability of real-time systems with a set of applications that are scheduled hierarchically. In this framework, the resource demand of a single task is represented as a demand bound function **dbf** [1] $w \in W$, and a scheduling component has as input the set of demand bound functions of all tasks that are scheduled by this component. Depending on the associated scheduling strategy, a scheduling component then determines the total demand to schedule all tasks and expresses this again as a demand bound function on its output. Scheduling components can then be composed hierarchically, and the complete system is schedulable if the demand of the scheduling component at the top of the hierarchy can be fulfilled by a dedicated resource. In the context of real-time interfaces, the scheduling components of this framework can be interpreted as abstract components (X, Y, T, Ψ), with a set of inputs $X : x \in W$ and outputs $Y : y \in W$. The transfer function $T(X)$ is applied to compute the outgoing demand bound function, and the predicate $\Psi(X)$ expresses the constraint that a component must be schedulable by a dedicated resource. It can be shown that these abstract components are monotone under the partial order defined as $w \geq \tilde{w} \Leftrightarrow w(t) \geq \tilde{w}(t), \forall t \geq 0$. Based on these abstract components, adaptive real-time interfaces can then be determined for this scheduling framework following the steps described in this paper.

5.3 Symta/S

In [12], Richter et al. propose a compositional approach to extend the concepts of classical scheduling theory to heterogeneous distributed systems. In this approach, every single processor or communication link is represented as a component that is analyzed locally. To interconnect the various components, the method relies on a set of standard event arrival patterns that are described as a tuple consisting of a period $p \in P$, a jitter $j \in J$ and a minimum event inter-arrival distance $d \in D$. Based on the arrival patterns of the incoming event streams and on the scheduling policy of the component, the appropriate analysis technique is chosen to compute the worst-case response time of every incoming event stream, and to compute the arrival patterns of the outgoing event streams that will trigger succeeding components. In the context of real-time interfaces, the components

of this framework can be interpreted as abstract components (X, Y, T, Ψ), with a set of inputs $X : x = (p, j, d) \in P \times J \times D$ and outputs $Y : y = (p, j, d) \in P \times J \times D$. The transfer function $T(X)$ is applied to compute the outgoing arrival patterns of event streams, and the predicate $\Psi(X)$ expresses the constraints on the maximum allowable response time for every event stream as well as the schedulability of the total component. It can be shown that these abstract components are monotone under the partial order defined as

$$(p, j, d) \geq (\tilde{p}, \tilde{j}, \tilde{d}) \quad \Leftrightarrow$$
$$\min\left\{\left\lceil\frac{\Delta + j}{p}\right\rceil, \left\lceil\frac{\Delta}{d}\right\rceil\right\} \geq \min\left\{\left\lceil\frac{\Delta + \tilde{j}}{\tilde{p}}\right\rceil, \left\lceil\frac{\Delta}{\tilde{d}}\right\rceil\right\}, \forall\Delta \geq 0$$

Based on these abstract components, adaptive real-time interfaces can then be determined for this compositional framework following the steps described in this paper.

5.4 Modular Performance Analysis

In [3], an alternative approach for modular performance analysis of real-time embedded system is proposed. In this approach, every task of a system is represented as a component that is analyzed locally. The method is based on arrival curves $\alpha(\Delta) \in \mathcal{A}$ as a generic event stream model [4] and on service curves $\beta(\Delta) \in \mathcal{B}$ as a generic resource model. Based on the task processing semantics, a task component relates incoming arrival and service curves that model the input event stream and the available resources to outgoing arrival and service curves that model the output event stream as well as the remaining resources, and analysis methods allow to compute delay bounds and buffer requirements at the task component. Task components are interconnected via their arrival curve inputs and outputs to reflect the flow of data in a system and via their service curve inputs and outputs to reflect the chosen scheduling policy in a system. In the context of real-time interfaces, task components can also be interpreted as abstract components (X, Y, T, Ψ), with a set of inputs $X : x \in \mathcal{A} \cup \mathcal{B}$ and outputs $Y : y \in \mathcal{A} \cup \mathcal{B}$. The transfer function $T(X)$ equals the internal component relations that are determined according to the processing semantics, and the predicate $\Psi(X)$ expresses the constraints on the maximum allowable delay or buffer requirements. It can again be shown that these abstract components are monotone under the partial order defined as $\alpha \geq \tilde{\alpha} \Leftrightarrow \alpha(\Delta) \geq \tilde{\alpha}(\Delta), \forall\Delta \geq 0$ and $\beta \geq \tilde{\beta} \Leftrightarrow \beta(\Delta) \geq \tilde{\beta}(\Delta), \forall\Delta \geq 0$. Based on these abstract components, adaptive real-time interfaces can again be determined following the steps described in this paper. Initial adaptive real-time interfaces for a subset of the modular performance analysis framework are already presented in [17] and [18].

6. CONCLUDING REMARKS

The paper presents real-time interfaces that enable compositional analysis of hierarchical and distributed real-time systems. In addition, the concept of adaptive interfaces is formally defined. It allows us to solve design and synthesis problems as the requirements and constraints can be dynamically propagated through the system. In addition, the adaptivity leads to tighter performance results as the properties of the components adapt to the current needs. Moreover, the adaptive interfaces could be implemented in the run-time system which leads to adaptive system behavior that depends on the current requests from the environment.

It should be mentioned that we use a transformational approach to real-time interfaces. On the other hand, this does not prevent us from modeling complex internal behavior of components or non-determinism in streams or components. For example, it has been shown in [16] that stateful and non-deterministic behavior of components can be dealt with. In addition, the abstraction used in Section 5.1 describes non-deterministic event streams and one could even use statistical stream models.

7. REFERENCES

[1] S. K. Baruah, *Dynamic- and static-priority scheduling of recurring real-time tasks*, Real-Time Systems **24** (2003), no. 1, 93–128.

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

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

[4] R.L. Cruz, *A calculus for network delay*, IEEE Trans. Information Theory **37** (1991), no. 1, 114–141.

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

[6] ———, *Interface theories for component-based design*, EMSOFT 01: Embedded Software, Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp. 148–165.

[7] ———, *Interface-based design*, To appear in the Proceedings of the 2004 Marktoberdorf Summer School, Kluwer, 2005.

[8] L. de Alfaro, T.A. Henzinger, and M. Stoelinga, *Timed interfaces*, EMSOFT 02: Embedded Software, Lecture Notes in Computer Science 2491, Springer-Verlag, 2002, pp. 108–122.

[9] T. A. Henzinger and S. Matic, *An interface algebra for real-time components*, Proceedings of the 12th Annual Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE Computer Society Pres, 2006.

[10] J.Y. Le Boudec and P. Thiran, *Network calculus - a theory of deterministic queuing systems for the internet*, LNCS 2050, Springer Verlag, 2001.

[11] A.K. Mok and Z.X. Feng, *Towards compositionality in real-time resource partitioning based on regularity bounds*, Proceedings of RTSS 2001, IEEE Computer Society Pres, 2001, pp. 129–138.

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

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

[14] ———, *Compositional Real-Time Scheduling Framework*, Proceedings of the Real-Time Systems Symposium (RTSS), IEEE Press, 2004, pp. 57–67.

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

[16] E. Wandeler and L. Thiele, *Abstracting functionality for modular performance analysis of hard real-time systems*, Asia South Pacific Design Automation Conference (ASP-DAC), 2005, pp. 697–702.

[17] ———, *Real-Time Interfaces for Interface-Based Design of Real-Time Systems with Fixed Priority Scheduling*, Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05), IEEE Press, 2005, pp. 80–89.

[18] ———, *Interface-based design of real-time systems with hierarchical scheduling*, Proceedings of the 12th Annual Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE Computer Society Pres, 2006.

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