Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems

Ingo Stierand*, Philipp Reinkemeier†, Tayfun Gezgin†, Purandar Bhaduri‡

*University of Oldenburg, Germany
stierand@informatik.uni-oldenburg.de

†OFFIS, Germany
{gezgin|reinkemeier}@offis.de

‡IIT Guwahati, India
pbhaduri@iitg.ernet.in

***Abstract*—A notion of interfaces based on regular languages for modelling and verification of real-time scheduling constraints was proposed in [5]. This initial notion considers task sets running on single resources, and simple deadline requirements. We extend the approach to enable support for complex task models running on systems with multiple resources. We show that this extension preserves all properties of the original notion. In addition, this extension gives rise to the application of our interfaces in the design of more complex systems, where components can be spread over distributed architectures.**

The work is complemented by an initial implementation that performs scheduling analysis for a relevant class of real-time interfaces. It actually constructs an interface for a system model if it satisfies a set of given real-time requirements.

I. Introduction

The design of safety-critical systems calls for rigorous formal design and verification methods as malfunctions might lead to catastrophic results. Recently, contract based design has received renewed attention in this area. New results [4] enable contract based design to solve important issues such as compositional reasoning, allowing us to construct even large systems from a set of well-designed components. Contract based design also paves the way for incremental design methods, where components are refined successively during the design process, starting from the initial high-level specification down to the final implementation. In [4] it is also discussed how contract based design can be applied in the deployment and mapping phase of the design process. Following the methodology advocated in *Platform Based Design* [19] a set of application contracts $\mathcal{C}$ is related to a set of platform contracts $\mathcal{P}$ by their conjunction $\mathcal{C} \wedge \mathcal{P}$. Obtaining a valid deployment then amounts to finding an execution platform $\mathcal{P}$ and a mapping of the application components on the platform components such that $\mathcal{C} \wedge \mathcal{P}$ is consistent.

In the context of contract based design, various concrete theories exist (assume/guarantee contracts, interface theories, timed interface theories, probabilistic interface theories, etc.), which are extensively discussed in [4]. However, the target hardware is considered in a rather abstract way, if at all. Interferences between different applications resulting from their deployment to the same platform component are not explicitly addressed by these theories. In other words, the methodological question of how to obtain an implementation of an application on a target platform is left open.

On the other hand real-time scheduling analysis [12] is an important building block in the deployment and mapping phase. It considers applications when they are deployed to processors, buses and other devices, thus exhibiting an explicit notion of a platform. Much work on real-time scheduling analysis has been done, with increasing expressive power along the evolution of analysis methods. It is impossible to give even a rough overview here, and we refer to [20] for a survey. Further, there has been a growing interest to relate compositional real-time scheduling analysis frameworks with interface-based design [11], [24], [22], [25] – as described in [4], interface-based design can be seen as an instantiation of the theory of contracts. These studies define interface theories for components abstracting the resource requirement of a component by means of demand functions. For example in [24], [22], [25] a notion of real-time interface is presented based on the Real-Time Calculus and the underlying principles of interface-based design [8].

The notion of real-time interfaces proposed in [5] is a complementary approach for modelling systems at the platform level when the system is implemented on a target processor. The formalism is based on the control interfaces proposed by Weiss and Alur [26], [1], [2]. It assumes a discrete time model, where time is divided into slots of pre-defined equal length. All scheduling related events, such as task arrivals, completions and preemptions, take place at these discrete time points. A schedule on a processor is described by an ω-word that describes the sequence of tasks that run in the discrete time slots. An ω-regular language describes the set of legal schedules.

These interfaces have been related to contracts allowing us to reason about whether a given interface – treated as implementation – satisfies a set of contracts. Moreover, the interface formalism is equipped with well-defined composition and refinement operations, which paves the way to a consistent application of contract based design methods down to the final implementation, explicitly addressing interferences between applications due to their deployment on a common target platform.

The expressiveness of the present interface formalism is still limited. Firstly, it does not capture task activations and completions directly but task executions only. Relating interfaces with

This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center 'Automatic Verification and Analysis of Complex Systems' (SFB/TR 14 AVACS).

task activations and completions, such as defined by a contract, requires a mapping function introducing inevitable ambiguity. This makes modelling of more complex systems such as with execution dependencies difficult, if not impossible. The second drawback is the restriction to single resources.

The present paper extends the interface notion, allowing us to express task activations and completions directly in the interface language. Similar to assume-guarantee interfaces [11], we equip our interfaces with input and output ports characterising interface behaviour at component boundaries. Along with this we add the ability to define interfaces on multiple resources. In addition to preserving all algebraic properties of the original notion, the extended one embeds consistently into contract-based design.

The paper presents an initial attempt at constructing interfaces that satisfy a given set of real-time requirements expressed as contracts. It is similar to other computational real-time scheduling analysis approaches based on model-checking, where the system is represented as a state-transition-system. The DREAM framework [13] for example transforms real-time models to the model-checking tool UPPAAL [7] for verification. The TIMES tool [3] also employs UPPAAL, but seems to access UPPAAL's internal state space in order to optimise analysis of scheduling specific properties. The framework FUNSTATE [21] exploits a symbolic representation of finite state machines called Interval Decision Diagrams, which enables efficient state-space storage and operations.

As interfaces are defined by ω-languages, the approach constructs the transition system in terms of a finite state machine (FSM). Along this process it is checked whether a given set of timing requirements is satisfied. If so, the resulting FSM represents the interface of the system that conforms to the given contracts by construction. While the present approach performs holistic (i.e., non-compositional) scheduling analysis we consider it as an important building block towards support for other key design steps such as incremental design and compositional analysis.

Section II introduces the extended real-time interface notion and the basic definitions for contracts used throughout the paper. It also defines a class of real-time models from which timing contracts and interfaces will be derived. Section III presents the approach for state space construction for a given model, and Section IV presents an evaluation of the implementation by a set of benchmarks. Section V discusses further work and concludes the paper.

II. INTERFACES, CONTRACTS AND SYSTEM MODEL

A. Real-Time Interfaces

The notion of real-time interfaces defined in [5] allows us to reason about real-time components that are executed on a single resource such as a processing node or communication medium. Each component consists of a set of tasks. A real-time interface of a component specifies the set of legal schedules when it is executed on the resource. To this end, time is divided into discrete slots of some fixed duration. The real-time interface of a component then is an ω-language containing only legal schedules of the component, i.e., those satisfying its requirements. For example, consider a component with two tasks τ_1 and τ_2, which are scheduled on a single resource, as shown in Figure 2 for component $CPU1$. A schedule for this component can be described by an infinite word over the alphabet $\{0, \tau_1, \tau_2\}$, where 0 means the resource is idle during the slot, and τ_1 and τ_2 means the corresponding task is running.

Example 1: Suppose that task τ_1 in Figure 2 is a periodic task with period $p = 5$ and an execution time $c = 3$. The language of its interface I_{τ_1} can be described by the following regular expression: $L_{\tau_1} = 0^{<5}[\tau_1^3 \,|||\, 0^2]^\omega$, where $u \,|||\, v$ denotes all possible interleavings of the finite words u and v. That means, a schedule is legal for interface I_{τ_1}, as long as it provides 3 slots during a time interval of length 5.

Observe, that interface I_{τ_1} captures an assumption about the activation pattern of task τ_1. The part $0^{<5}$ of the regular expression represents all possible phasings of the initial task activation. This correlates to the formalism of event streams, which is a well-known representation of task activation patterns in real-time systems (cf. [17]) by lower and upper arrival curves $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$.

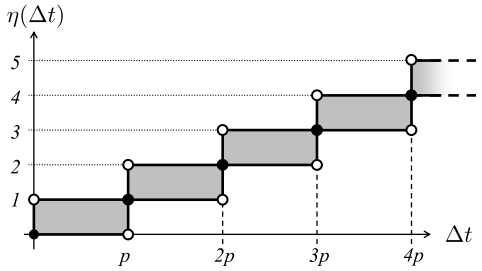


Fig. 1: Arrival curves of periodic events

While this notion of real-time interfaces provides desired properties such as composability and refinement, it lacks two important features. Firstly, as the interface does not capture task activations and completions but task executions only, modelling more elaborate task systems such as with task dependencies becomes complex, if not impossible. In order to relate interfaces with contracts that characterise task activations and completions, [5] defines a function α that maps such events to possible tasks executions in terms of interface languages. Such mapping functions however do not solve this issue properly, because in general no unique inverse mapping exists. Secondly, this notion does not allow us to define interfaces for multiple resources.

Hence we consider interfaces defined over tuples of symbols. A component has a set $P = P_{in} \uplus P_{out}$ of input and output ports. Symbols occurring at the individual ports represent activation and completion events for the tasks that are connected to the corresponding ports. The events that may be observed at port $p \in P$ are characterised by the alphabet Σ_p, and we define $\Sigma_P = \Sigma_{p_1} \times \ldots \times \Sigma_{p_n}$. As task activations and completions might not occur at each time step, we assume a special symbol $\perp$ denoting that no event occurs.

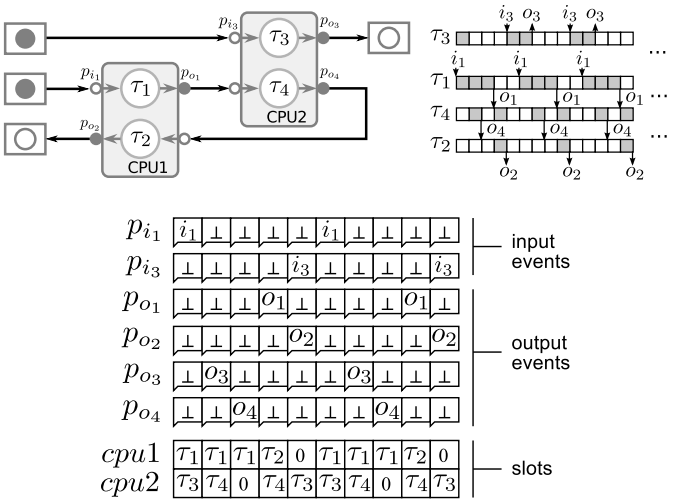


Fig. 2: System example (top) and exemplary trace-extract (bottom) of the corresponding interface.

While [5] defines interfaces by languages over a set $\mathcal{T}$ of tasks that occupy slots of a single resource, we extend this to sets R of resources that are running in parallel. To each resource $r \in R$ a set of tasks is allocated, which is represented by the alphabet Σ_r.

Definition 1: An *interface* is a tuple $I_K = (K, \Sigma_K, L_K)$ where $K = P \cup R$ is an index set of ports P and resources R where:

- For $k \in P, \Sigma_k$ is the set of events that may occur at port k. $\perp \in \Sigma_k$ means "no event".
- For $k \in R, \Sigma_k$ is the set of tasks that run on resource k. $0 \in \Sigma_k$ means "slot not used".

and $\Sigma_K = \prod_{k \in K} \Sigma_k$, $L_K \subseteq \Sigma_K^\omega$. ◇

Example 2: Focus on the example system depicted on the left hand side of Figure 2. Suppose that task τ_1 is a periodic task with period $p = 5$ and an execution time $c = 3$ just like in Example 1. Task τ_3 is also a periodic task with $p = 5$ and $c = 2$. Further, task τ_4 is dependent on τ_1, i. e., is activated by τ_1, and has an execution time $c = 2$. Task τ_2 is dependent on τ_4 and has an execution time $c = 1$. Now assume both CPUs are scheduled using a fixed priority policy with preemption, where tasks τ_1 and τ_3 have high priority on their respective CPUs. The delay of the task-chain $\tau_1 \to \tau_4 \to \tau_2$ processing the periodic event stream depends on the activation-pattern of τ_3 and its execution time. This is illustrated on the right hand side of Figure 2. Once τ_1 completes its execution it activates (via its output port p_{o_1}) τ_4, which in turn might be preempted by τ_3. Finally, τ_2, activated by τ_4, could be preempted by a subsequent instance of τ_1 resulting from another event i_1 of the periodic event stream. The interface of this system is $I_K = (K, \Sigma_K, L_K)$, with $K = P \cup R$, $P = \{p_{i_1}, p_{i_3}, p_{o_1}, \ldots, p_{o_4}\}$ and $R = \{cpu1, cpu2\}$. For $p_{i_j} \in P$ we have $\Sigma_{p_{i_j}} = \{i_j, \perp\}$, for $p_{o_j} \in P$ we have $\Sigma_{p_{o_j}} = \{o_j, \perp\}$, for $cpu1$ we have $\Sigma_{cpu1} = \{\tau_1, \tau_2, 0\}$ and for $cpu2$ we have $\Sigma_{cpu2} = \{\tau_3, \tau_4, 0\}$. An excerpt of a possible trace in L_K is shown in Figure 2, which corresponds to the previously discussed scheduling scenario. Intuitively, each port in a system has its own event-tape in the interface, and so does each resource. Note, that we omitted input ports connected to some output port. This is because we define a connection between tasks by a unification of their ports to denote a synchronisation of the behaviour.

The key to dealing with interfaces having different alphabets is a projection operation. For alphabet $\Sigma_\mathcal{T}$ and language L of a (simple) interface I, and $\Sigma_{\mathcal{T}'} \subseteq \Sigma_\mathcal{T}$, we consider its *projection* $proj(\Sigma_\mathcal{T}, \Sigma_{\mathcal{T}'})(L)$ to $\Sigma_{\mathcal{T}'}$, which is the unique extension of the function $\Sigma_\mathcal{T} \to \Sigma_{\mathcal{T}'}$ that is identity on the elements of $\Sigma_{\mathcal{T}'}$ and maps every element of $\Sigma_\mathcal{T} \setminus \Sigma_{\mathcal{T}'}$ to 0. We will also need the *inverse projection* $proj^{-1}(\Sigma_{\mathcal{T}''}, \Sigma_\mathcal{T})(L)$, for $\Sigma_{\mathcal{T}''} \supseteq \Sigma_\mathcal{T}$, which is the language over $\Sigma_{\mathcal{T}''}$ whose words projected to $\Sigma_\mathcal{T}$ belong to L.

Notation: For $f : X \to Y$, $A \subseteq X$ and $B \subseteq Y$, we write $f(A)$ for the direct image $\{f(a) \mid a \in A\}$ and $f^{-1}(B)$ for the inverse image $\{x \in X \mid f(x) \in B\}$.

For interfaces that are defined over alphabets of the form $\Sigma_K = \Sigma_{k_1} \times \ldots \times \Sigma_{k_n}$, projection becomes more involved. First, projection must be performed component-wise, i.e., for each k_i individually. Secondly, we have to consider interfaces that are defined over different index sets. To this end, we define *normalisation* operations. Let K and $K' \subseteq K$ be index sets. For an alphabet $\Sigma_{K'}$ we define $\Sigma_{K' \to K} = \prod_{k \in K} \Sigma'_k$ where $\Sigma'_k = \Sigma_k$ if $k \in K'$, and $\{0\}$ otherwise. For an alphabet Δ_K we define $\Delta_K|_{K'} = \prod_{k \in K'} \Delta_k$.

Definition 2: Let $N = \{1, ..., n\}$, and let $\Sigma = \Sigma_1 \times ... \times \Sigma_n$ and $\Delta = \Delta_1 \times ... \times \Delta_n$ be alphabets with $\Sigma_i \subseteq \Delta_i$ for $i = 1, ..., n$. Define projection function $proj(\Delta, \Sigma) : \Delta^\omega \to \Sigma^\omega$ by the unique extension of the function $proj(\Delta, \Sigma) : \Delta \to \Sigma$ where $proj(\Delta, \Sigma)(\delta_1, ..., \delta_n) = (\sigma_1, ..., \sigma_n)$ such that $\sigma_i = \delta_i$ if $\delta_i \in \Sigma_i$, and 0 otherwise.

For $M = \{i_1, \ldots, i_m\} \subseteq N$ and $\Sigma' = \Sigma_{i_1} \times ... \times \Sigma_{i_m}$ we define $proj(\Delta, \Sigma')(L) := proj(\Delta|_M, \Sigma')(L|_M)$. ◇

In other words, if $\Sigma_i \subseteq \Delta_i$ then projecting a word over the larger alphabet Δ_i into a word over the smaller alphabet Σ_i will map any symbol from Δ_i not belonging to Σ_i to 0; symbols that belong to Σ_i will be mapped to themselves. The projection of a word over Σ then projects all elements i simultaneously. Taking the inverse projection of a word over Σ_i will result in a set of words where any 0 in the word will be replaced by all the letters in Δ_i which are not in Σ_i. The inverse projection of a word over Σ results in a set of words with all combinations of replacements for the individual elements.

We define a composition operation to obtain the set of schedules when two components are executed together:

Definition 3: Given two interfaces $I_1 = (K_1, \Sigma_{K_1}, L_{K_1})$ and $I_2 = (K_2, \Sigma_{K_2}, L_{K_2})$, the *parallel composition* $I_1 \parallel I_2$ is the interface (K, Σ_K, L_K), where

- $K = K_1 \cup K_2$,
- $\Sigma_K = \prod_{k \in K} (\Sigma_{K_1 \to K}|_k \cup \Sigma_{K_2 \to K}|_k)$
- $L_K = proj^{-1}(\Sigma_K, \Sigma_{K_1})(L_{K_1})$
  $\cap\ proj^{-1}(\Sigma_K, \Sigma_{K_2})(L_{K_2})$ ◇

The intuition of this definition is that a schedule is legal for $I_1 \parallel I_2$ if its restriction to resources R_1 and the port set P_1 of interface I_1 is legal in I_1, and similarly for interface I_2. That means tasks of an interface are allowed to run in a slot of resource $r \in R$ when r is idle in the other interface, i. e., the slot is not used in that other interface. Additionally, the projection operation controls which ports of I_1 and I_2 shall be related, i. e., which events shall be synchronised in the composition. This intuition is illustrated in Figure 2. Ports connected in the system are unified in the corresponding interface (for example port p_{o_1}), which means the same behavior can be observed at connected ports.

Definition 4: Given two interfaces $I = (K, \Sigma_K, L_K)$ and $I' = (K', \Sigma_{K'}, L_{K'})$, we say I' *refines* I, $I' \preceq I$, if and only if $K' \supseteq K$, $\Sigma_{K'} \supseteq \Sigma_{K \to K'}$ and $proj(\Sigma_{K'}, \Sigma_K)(L_{K'}) \subseteq L_K$. $\diamond$

The intuition of this definition is that all schedules legal in I' are (modulo projection) also legal schedules in I, and I' is able to schedule more tasks in the gaps left by schedules in I. Note that, if I and I' are defined over the same index sets and alphabets, refinement becomes simple language inclusion: $L_{K'} \subseteq L_K$.

The following lemmas provide useful properties of the real-time interface framework.

Lemma 1: Parallel composition of interfaces is associative and commutative. $\diamond$

An associative and commutative composition operation guarantees that composable interfaces may be assembled together in any order.

Lemma 2: Refinement of interfaces is a partial order. $\diamond$

As refinement is a partial order, it is ensured that: If interface $I' \preceq I$, then for any interface $I'' \preceq I'$ it holds that $I'' \preceq I$. That means interfaces can be refined iteratively.

Lemma 3: Refinement is compositional. That means $I' \preceq I$ implies $I' \parallel J \preceq I \parallel J$. $\diamond$

B. Adding Contracts

As in [5] we equip our real-time interfaces with a notion of *contracts*. Contracts are pairs (A, G) where A is an *assumption* about the environment of a component, and G is the *guarantee* that the component offers to its environment [4]. For real-time interfaces, both assumptions and guarantees will talk about bounds on the frequency of task arrivals and time to completions. In addition, they can capture the dependencies between tasks, for example, by stating that "task 2 is triggered whenever task 1 completes".

Both, the assumptions A and the guarantees G, consist of task release (or arrival) times as well as task finishing (or completion) times. These are again modelled using ω-regular languages, but now the semantics is about the behaviour observed at the ports P of a component. An ω-language of a contract is defined over the set Σ_P of events, and corresponds to time instants when either nothing happens (modelled by $\perp$), a task arrives (modelled by an event at the input port of the task) or finishes execution (modelled by an event at an output port). The contract (A, G), where $A \subseteq \Sigma_P^\omega$ and $G \subseteq \Sigma_P^\omega$ specifies promises on the arrival and finishing times of a set of tasks, given the assumptions on the arrival and finishing times of the same set of tasks. A dependency between tasks, such as task τ_i triggers task τ_j, is captured by the occurrence of an event at the port that connects the two tasks. When we compose components it becomes important to care about which ports contracts talk about. Hence we define a contract over a set of ports as a tuple $C = (P, \Sigma_P, A, G)$ where $A, G \subseteq \Sigma_P^\omega$.

In [5] the relation between contracts and interfaces is provided by a map α that translates the assumptions and guarantees into interface languages. Suppose a periodic task i with period p_i, zero phasing, and relative deadline d_i. Assume that $d_i \leq p_i$. This is captured by the contract (A, G) where $A = (\bigcup_{s+t=p_i-1} a_i(0^s) f_i(0^t))^\omega$ and $G = (0^\star a_i 0^{\leq d_i - 1} f_i)^\omega$. The events a_i and f_i represent the activation and completion of task i. Suppose task i has execution time of c_i on the processor. The translation of A into a set of task executions is given by $\alpha(A) = (i^{c_i} \,|||\, 0^{p_i - c_i})^\omega$ and that of G by $\alpha(G) = (0^\star (i^{c_i} \,|||\, 0^{d_i - c_i}))^\omega$.

With this translation, checking whether an interface complies to a contract reduces to language inclusion with respect to the mapping α. Indeed, the mapping involves an abstraction of the behaviour, which causes problems for more complex scenarios, as the mapping induces more behaviour in the interface than intended. With the revised definition of interfaces, these issues in the relation with contracts disappear:

Definition 5: [4] Let $C = (P, \Sigma_P, A, G)$ be a contract. An implementation M of the contract *satisfies* C, written $M \vDash C$, if and only if $M|_P \cap A \subseteq G$. Here M, A and G are all sets of traces (sequences). $\diamond$

Considering interfaces as implementations of contracts, we get the following relation. An interface $I_K = (K, \Sigma_K, L_K)$ satisfies a contract $C = (P, \Sigma_P, A, G)$ if L_K satisfies C.

We define a parallel composition of contracts that is consistent with the definition in [4]. However, in order to reason about contracts over different port sets, it is necessary to equalise the alphabets of the involved assertions. This is done exactly as for interfaces:

Definition 6: Let $C_1 = (P_1, \Sigma_{P_1}, A_1, G_1)$ and $C_2 = (P_2, \Sigma_{P_2}, A_2, G_2)$ be contracts. The parallel composition $C_1 \parallel C_2$ is the contract $C = (P, \Sigma_P, A, G)$ where $P = P_1 \cup P_2$, $\Sigma_P = \prod_{p \in P} (\Sigma_{P_1 \to P}|_p \cup \Sigma_{P_2 \to P}|_p)$, and

$$A = (A_1' \cap A_2') \cup \neg(G_1' \cap G_2'),$$
$$G = G_1' \cap G_2',$$

$A_i' = proj^{-1}(\Sigma_P, \Sigma_{P_i})(A_i)$ and $G_i' = proj^{-1}(\Sigma_P, \Sigma_{P_i})(G_i)$. $\neg X$ denotes the complement of X. $\diamond$

We conclude the section by revisiting the proposition about compositionality of contract satisfaction. The following lemma states that the satisfaction relation between implementations and contracts is also compositional for the revised notion of interfaces.

Lemma 4: If the interfaces I_1 and I_2 satisfy contracts C_1 and C_2 respectively, then $I_1 \parallel I_2$ satisfies $C_1 \parallel C_2$. $\diamond$

C. System Model

We now instantiate the framework for a relevant class of models in the context of real-time systems. We first characterise this class, and then define how to obtain contracts and interfaces for a given model.

Applications are modelled by networks of tasks, which define the atomic functional units to be executed on a target architecture. Tasks are also used to model messages that are transmitted over a communication medium [23]. Task invocations (execution requests) are modelled by events occurring at particular ports, and the same holds for task completions. Task dependencies are modelled by the unification of their ports.

Tasks without dependencies are connected to event sources that represent task activations from the environment. For each event source, we define an activation pattern that characterises event occurrences. Following the approach for assumed activation patterns of real-time interfaces outlined in Example 1, we consider activation patterns conforming to the formalism of event streams. We define a particular class of event streams characterised by tuples $(\Sigma, \rho^-, \rho^+, j)$ where $\rho^-, \rho^+ \in \mathbb{N}^+$, and $j \in \mathbb{N}$. It defines an interval $[\rho^-, \rho^+]$ that determines the minimal and maximal inter-arrival time between individual events from the set Σ. Additionally, each event might be further delayed up to a maximum jitter j.

Definition 7: A *task network* is a tuple $TN = (\Sigma, P, \Phi, \mathcal{T})$ where:

- Σ is a finite set of events,
- P is a finite set of ports. We define $\Sigma(p) \subseteq \Sigma$ as the set of events that can be observed at port $p \in P$, and $\Sigma(Q) = \bigcup_{p \in Q} \Sigma(p)$ for $Q \subseteq P$.
- Φ is a finite set of event sources $\phi = (P_\phi, \rho^-, \rho^+, j)$ where $P_\phi \subseteq P$ is a set of output ports, $P_\phi \neq \emptyset$, and all ports share the same event set, i.e., $\forall p \in P_\phi : \ \Sigma(p) = \Sigma(P_\phi)$. $(\Sigma(P_\phi), \rho^-, \rho^+, j)$ forms an event pattern.
- $\mathcal{T}$ is a finite set of tasks $\tau = (p_\tau^I, \Gamma, P_\tau^O)$ where $p_\tau^I \in P$ is an input port, $\emptyset \neq P_\tau^O \subseteq P$ a set of output ports, and

$$\Gamma = \Sigma(p_\tau^I) \to \Psi$$
$$\Psi = \bigcup\nolimits_{\emptyset \neq Q \subseteq P_\tau^O} \{\psi : Q \to (\Sigma(Q) \times \mathbb{N}^+ \times \mathbb{N}^+) \mid \psi(p_o) = (\sigma, \delta^-, \delta^+) \implies \sigma \in \Sigma(p_o)\}$$

  where Γ maps input events arriving at the input port p_τ^I to an output specification $\psi \in \Psi$ that maps output ports to output events and execution intervals. $\diamond$

A set Ψ defined for a task τ denotes all non-empty subsets of output ports of τ in combination with output events and execution intervals $[\delta^-, \delta^+]$. The input/output function of a task hence allows sending events to any combination of output ports, depending on the input event received at its input port. Bounds for execution intervals are assumed to be known.

We require task networks to be well-formed, i.e., input and output ports of the individual tasks must be disjoint. We also require that the composition is closed, i.e., where each task is either connected to a preceding task or to an event source. Because tasks have a single input port, only tree-shaped task networks can be defined. This is not a general restriction but to keep the definitions simple.

The following definition provides a notion of architecture represented by a set of resources R, and an allocation of a task network TN to the resources:

Definition 8: A *real-time model* is a tuple $\mathcal{A} = (TN, R, \Xi)$ where:

- TN is a task network,
- R is a set of resources, where $r \in R$ is defined by a scheduling strategy $sch : R \to \{FPS, \ldots\}$,
- $\Xi : \mathcal{T} \to R$ is an allocation function that assigns a resource $r \in R$ to every task $\tau \in \mathcal{T}$. $\diamond$

Observe that Ξ induces a set of input and output ports $P_r \subseteq P$ for each resource $r \in R$, which contains the input and output ports of all tasks mapped to r. Note that the set of scheduling strategies is deliberately abstract. We like to point out that the approach can cope with a broad range of scheduling schemes.

We consider a real-time model as a system characterisation for which we create contracts and interfaces. Given a task network $TN = (\Sigma, P, \Phi, \mathcal{T})$ we obtain the corresponding contract as follows. As the event sources in Φ characterise the activation of the application by the environment, their behaviour serves as the assumption of the contract specifying the application. As ω-languages correspond to finite state machines, the language of a single event source can be characterised by the corresponding automaton. An example for a periodic event source is shown in Figure 4a. Event sources are independent, and hence the assumption for a set of event sources is obtained by the product of the respective languages. If $L_A(\phi)$ is the language of event source ϕ, the assumption is $A = \prod_{\phi \in \Phi} L_A(\phi) = \{(\sigma_{0,1}, \ldots \sigma_{0,n})(\sigma_{1,1}, \ldots \sigma_{1,n}) \ldots \mid \sigma_{0,i}\sigma_{1,i} \ldots \in L_A(\phi_i)\}$.

The guarantee of the contract defines a set of *deadlines*. To keep the discussion simple we consider local deadlines only. For each task $\tau \in \mathcal{T}$ we define a mapping $D : \Sigma(p_\tau^I) \to \mathbb{N}^+$ that assigns to each possible activation event of τ the maximal allowed time to completion. Suppose for simplicity the case where the deadline of task τ with a single output port is less than its minimal inter-arrival time for every activation. Then we can derive the guarantee from D and the function Γ_τ. For a task with deadline smaller than the minimal inter-arrival time it results in a language of the form $L_G(\tau) = [0^* \sigma_{p_\tau^I} 0^{<D(\sigma_{p_\tau^I})} \sigma_{p_\tau^O}]^\omega$. The guarantee of the model is obtained by $G = \bigcap_{\tau \in \mathcal{T}} proj^{-1}(\Sigma_P, \Sigma(P_\tau))(L_G(\tau))$.

The language of an interface subsumes all ports P and resources R of the model. It contains all valid schedules with respect to the underlying architecture and scheduling policies. The interface of a real-time model can be obtained in various ways, depending on the intended scenario and design process. For abstract interfaces, where no concrete scheduling is yet defined, mapping functions as discussed earlier can be exploited. The next section introduces an approach to construct the interface of a real-time model when scheduling policies and contracts are known.

III. Real-Time Analysis Approach

The remainder of this paper presents an approach to obtain initial tool support for the real-time interface formalism. We concentrate on the question of how to construct interfaces with respect to an underlying architecture and given scheduling policies. A complementary view to this aspect is the well-known area of real-time scheduling analysis [20], where it is checked whether a set of tasks can be scheduled on given hardware architecture without violating given (deadline) requirements.

In our setting, this is equivalent to finding an interface for a given real-time model and deadlines. As interfaces essentially are ω-languages, finite state machines (FSM) are an obvious formalism on which interface-based tools can be constructed. We construct for a given real-time model an FSM that represents all possible behaviour of the system with respect to the underlying architecture. Then it is checked whether this behaviour satisfies all given timing requirements, which are given in terms of contracts. If this holds, the resulting FSM represents an interface of the real-time model that satisfies the given contract.

The following presentation concentrates on the construction principles while omitting implementation details.

A. Finite State Transducers and System Composition

In the following we consider systems that are composed from sets of so-called *transducers*, which are finite state machines operating on distinct input and output ports. Transducers are a well-established formalism with Moore- and Mealy-machines [14], [15] as their most common representatives. We assume a global non-empty finite set $P = \{p_1, ..., p_n\}$ of ports, and alphabets Σ_p for all $p \in P$. We require a special symbol $\epsilon \in \Sigma_p$ for each $p \in P$. This symbol serves as a don't care indicator. In contrast to the earlier sections now all components of a system will be defined over the same alphabet $\Sigma_P = \Sigma_{p_1} \times \ldots \times \Sigma_{p_n}$. A component that does not act on a particular port shows only ϵ symbols on that port. Note that ϵ has the same meaning as the symbol 0 before, but ϵ is used in the following for better readability.

For $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Sigma_P$ and $Q \subseteq P$, we define *projection* of σ to Q as $\sigma|_{\epsilon Q} = (\widetilde{\sigma_1}, \ldots, \widetilde{\sigma_n})$, where $\widetilde{\sigma_i} = \sigma_i$ if $i \in Q$ and $\widetilde{\sigma_i} = \epsilon$ otherwise. We extend projection to sequences $\omega|_{\epsilon Q}$, and to languages $L|_{\epsilon Q}$. We also define $\Sigma_P|_{\epsilon Q} = \{\sigma|_{\epsilon Q} \mid \sigma \in \Sigma_P\}$.

We further define a composition operation on events. Let σ be an event from an alphabet Σ. Then the empty event ϵ is an identity element with respect to the composition operation $\otimes$, i.e., $\sigma \otimes \epsilon = \epsilon \otimes \sigma = \sigma$. In fact we consider $\sigma \otimes \sigma'$ only being defined if either σ or σ' is ϵ. Event composition is extended to events from the global alphabet as follows: Let σ and σ' be two events from Σ_P. We define $\sigma \otimes \sigma' = (\sigma_{p_1} \otimes \sigma'_{p_1}, \ldots, \sigma_{p_n} \otimes \sigma'_{p_n})$ as the pair-wise composition of σ and σ'. Again, $\sigma \otimes \sigma'$ is defined only if all $\sigma_{p_i} \otimes \sigma'_{p_i}$ are defined.

A *finite state transducer* (FST) is a tuple $F = (\Sigma_P, P_{in}, P_{out}, S, s_0, T, G)$ where

- $P_{in} \subseteq P$ is a set of input ports,
- $P_{out} \subseteq P$ is a set of output ports s.t. $P_{in} \cap P_{out} = \emptyset$,
- S is a finite set of states, and $s_0 \in S$ is the initial state,
- $T \subseteq S \times \Sigma_P|_{\epsilon P_{in}} \times S$ is a set of transitions,
- $G : S \to \Sigma_P|_{\epsilon P_{out}}$ is the output function of F.

We say F is closed if $P_{in} = \emptyset$, that is F has no input ports.

A run of F is an infinite sequence $s_0\sigma_0 s_1\sigma_1 \ldots$ such that $(s_i, \sigma_i|_{\epsilon P_{in}}, s_{i+1}) \in T$ and $\sigma_i = \sigma_i|_{\epsilon P_{in}} \otimes G(s_i)$ for all $i \in \mathbb{N}$. Each run $s_0\sigma_0 s_1\sigma_1 \ldots$ of F induces a trace $\sigma_0\sigma_1 \ldots$. The language $L(F)$ is the set of all possible traces of F.

For a subset $Q \subseteq P$, we define the FST $F|_{\epsilon Q} = (\Sigma_P, P_{in}{}', P_{out}{}', S, s_0, T', G')$ where $P_{in}{}' = P_{in} \cap Q$, $P_{out}{}' = P_{out} \cap Q$, and

$$(s, \sigma|_{\epsilon P_{in}{}'}, s') \in T' \iff (s, \sigma, s') \in T,$$
$$G'(s) = \sigma|_{\epsilon P_{out}{}'} \iff G(s) = \sigma.$$

A *system* is a set $\mathcal{S} = \{F_1, \ldots, F_m\}$ of FSTs such that for all $F_i = (\Sigma_P, P_{ini}, P_{outi}, S_i, s_{i,0}, T_i, G_i)$ and $F_j = (\Sigma_P, P_{inj}, P_{outj}, S_j, s_{j,0}, T_j, G_j)$ with $i \neq j$ the relation $P_{ini} \cap P_{inj} = P_{outi} \cap P_{outj} = \emptyset$ holds. Composition of $\mathcal{S}$ is the FST $F_{\mathcal{S}} = (\Sigma_P, P_{in}, P_{out}, S, s_0, T, G)$ where

- $P_{out} = \bigcup_{j \in \{1,\ldots,m\}} P_{outj}$
- $P_{in} = \bigcup_{j \in \{1,\ldots,m\}} P_{inj} \setminus P_{out}$
- $S = S_1 \times \ldots \times S_m$, with S_j being the states of FST F_j
- $s_0 = (s_{1,0}, \ldots, s_{m,0})$
- $G(s_1, \ldots, s_m) = G_1(s_1) \otimes \ldots \otimes G_m(s_m)$
- $((s_1, \ldots, s_m), \sigma_1 \otimes \ldots \otimes \sigma_m, (s'_1, \ldots, s'_m)) \in T \iff \forall j \in \{1, \ldots, m\} : (s_j, \sigma_j, s'_j) \in T_j \wedge \sigma_j|_{\epsilon P_{out}} = G(s_1, \ldots, s_m)|_{\epsilon P_{inj}}$

A system $\mathcal{S}$ is closed if $F_{\mathcal{S}}$ is closed. Intuitively, $F_{\mathcal{S}}$ synchronises the transitions of all FSTs of $\mathcal{S}$ with the last output of all FSTs. Output function G of the composed FST is well-defined as we require the output ports of the involved FSTs to be pair-wise disjoint. Hence the composed output is the output of each FST involved in the composition. Also the transition symbols $\sigma_1 \otimes \ldots \otimes \sigma_m$ are well-defined as we require the input ports of the involved FSTs to be pair-wise disjoint. Synchronisation of the individual FSTs is ensured by the requirement $\sigma_j|_{\epsilon P_{out}} = G(s_1, \ldots, s_m)|_{\epsilon P_{inj}}$. It states that the transition of an FST can be taken only if each input port of the FST is either not connected to an output port of another FST, or the output event of a connected FST, projected to the input port, matches the input event. In the former case the event is not restricted. Figure 3 depicts the relationship.

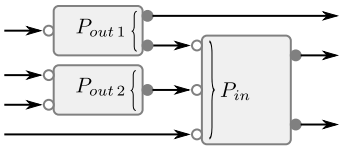


Fig. 3: Port Composition

A naive implementation of the composition operation requires a preliminary construction of the FSTs for all components. Since composition typically "cuts away" large portions of those local state spaces (actually all states that are not

reachable), preliminary construction of local state spaces for each component would result in large overheads. Hence, we aim at performing composition in an iterative process:

Algorithm 1 (Iterative FST Composition): Let $\mathcal{S}$ be a closed system. The FST $F_{\mathcal{S}}$ is constructed as follows:

1) Initially the state set of $F_{\mathcal{S}}$ is $S = \{(s_{1,0}, \ldots, s_{m,0})\}$.
2) We define set S_{ch} of *changed states*. Initially $S_{ch} = S$.
3) While $S_{ch} \neq \emptyset$, do
   a) Take a state $(s_1, \ldots, s_m) \in S_{ch}$, removing it from S_{ch}.
   b) Compute $G(s_1, \ldots, s_m) = G_1(s_1) \otimes \ldots \otimes G_m(s_m)$.
   c) For each F_i, given its current state s_i, determine successor states S'_i reachable by $G(s_1, \ldots, s_m)|_{\epsilon P_{in_i}}$.
   d) For each element $(s'_1, \ldots, s'_m) \in S'_1 \times ... \times S'_m$: Add $((s_1, \ldots, s_m), G(s_1, \ldots, s_m), (s'_1, \ldots, s'_m))$ to T. If $(s'_1, \ldots, s'_m) \notin S$, then add it to the sets S_{ch} and S.

B. FST Generators

Instead of defining complex components, such as a resource scheduler executing sets of real-time tasks, directly as FSTs, it is much more convenient to define them over states that represent sets of variables $v_1, \ldots, v_n$ from a finite domain $\mathcal{D} = \mathcal{D}_1 \times \ldots \mathcal{D}_n$. This allows us to define states that represent, for example, (bounded) integer values and buffer contents. Evolution of such a component is defined by two functions: $m : \mathcal{D} \times \Sigma_P|_{\epsilon P_{in}} \rightarrow 2^{\mathcal{D}}$ defines a set of possible successor states reachable within a single time step for a given state and an input event; and $g : \mathcal{D} \rightarrow \Sigma_P|_{\epsilon P_{out}}$ maps local states to output events. To generalise this idea, we define an FST generator as a tuple $M = (\Sigma_P, P_{in}, P_{out}, \mathcal{D}, v_0, m, g)$ where $v_0 \in \mathcal{D}$ is the initial local state. Every generator M uniquely identifies an FST $F_M = (\Sigma_P, P_{in}, P_{out}, S, s_0, T, G)$ where

- S such that $|S| = |\mathcal{D}|$, and $\nu : S \rightarrow \mathcal{D}$ is a bijective mapping where $\nu(s_0) = v_0$,
- $(s, \sigma, s') \in T \iff \nu(s') \in m(\nu(s), \sigma)$,
- $G(s) = \sigma \iff g(\nu(s)) = \sigma$.

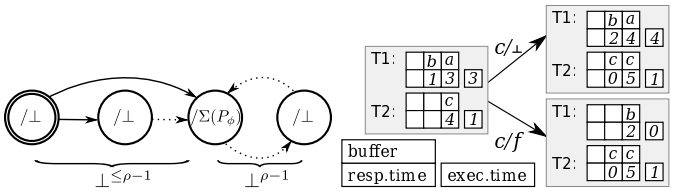
This allows us to directly apply FST generators to Algorithm 1. For each step of the algorithm, mapping ν identifies the corresponding state of the generator. Mapping m ensures that successor states and transitions can be obtained, and mapping g provides the output events for the respective states.

C. State Space Construction

Definition 9: Let $\mathcal{A} = (TN, R, \Xi)$ be a real-time model where $TN = (\Sigma, P, \Phi, \mathcal{T})$. We define the *real-time system* $\mathcal{S}_{\mathcal{A}} = (P, \Sigma_P, \mathcal{F}_{\mathcal{M}})$ where $\mathcal{F}_{\mathcal{M}} = \{F_M \mid M \in \mathcal{M}\}$, and:

- $\Sigma_P = \Sigma_{p_1} \times \ldots \times \Sigma_{p_n}$ such that $\Sigma_p = \Sigma(p) \cup \{\epsilon, \perp\}$ for all $p \in P$,
- $\mathcal{M} = \mathcal{M}_\Phi \cup \mathcal{M}_R$, where $\mathcal{M}_\Phi = \{M_\phi \mid \phi \in \Phi\}$ is a set of event source generators, and $\mathcal{M}_R = \{M_r \mid r \in R\}$ is a set of resource generators. $\diamond$

Recall that the task network TN is closed by definition. Hence, $\mathcal{S}_{\mathcal{A}}$ is also closed. The construction of FST generators for event sources is straightforward. It is in fact a discretised version of the automaton templates defined by Hendriks and Verhoef in [10]. The FST constructed for a simple periodic event source $\phi = (P_\phi, \rho, \rho, 0)$ is depicted in Figure 4a. The first event, which belongs to $\Sigma(P_\phi)$, is emitted non-deterministically within the first ρ steps. Afterwards they are emitted each ρ steps.



(a) FST for $\phi = (P_\phi, \rho, \rho, 0)$ (b) Resource State Example

Fig. 4: State representation for event sources and resources

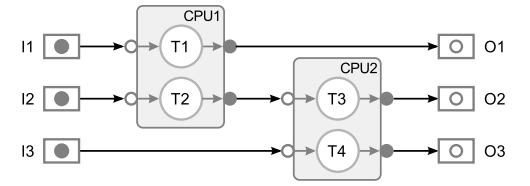
Figure 4b depicts the general encoding of a state of a resource generator. Each allocated task is represented by a data structure as shown at the bottom left of the figure. Activation events for the task are stored in a buffer together with a response time counter. In Figure 4b, task τ_1 has been activated by an event a, followed by an activation due to event b. An additional counter stores the elapsed execution time of the current activation. At each step all response time counters are incremented. Incoming activations are stored in the respective buffers, as shown for an activation of task τ_2 by event c. According to the scheduling strategy, the execution counter of the currently active task is incremented. In Figure 4b this is for task τ_1 as it has higher priority. Calculation of successor states includes the decision whether tasks are completed. In Figure 4b, task τ_1 has execution time between 4 and 6. Hence two successor states exist. The first possible outcome is that τ_1 is still executed. Otherwise it is finished, and a completion event f is sent.

This encoding allows for efficient calculation of local response times. As each resource state that is constructed during system composition is reachable, it is sufficient to traverse all states once only in order to obtain the worst-case response time for all tasks allocated to the resource. This can be done separately for each resource as resource states are kept local. Linking between the states of the composed FST and local resource states is obtained by the mappings ν_j. For a state $(s_1, \ldots, s_m)$ of the composed FST, the local state v_j of resource R_j is given by $\nu_j(s_j)$.

Calculation of end-to-end response times for task chains involves breadth-first search (BFS) on the composed system FST. At each of the states where a task in the chain is completed, the algorithm starts a BFS to find the longest path to completion of the successor task in the chain. Note that the longest path has length equal to the worst-case response time of the respective task.

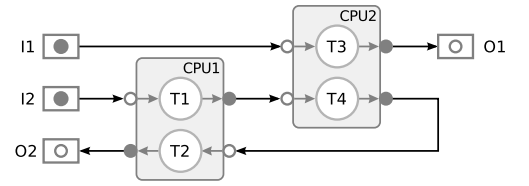
IV. Evaluation

To evaluate the feasibility of our approach we compare the analysis results of our implementation (in the following denoted RTANA$_2$) with other performance analysis frameworks, based on a set of benchmarks published in [16]. The authors studied the influence of different abstraction



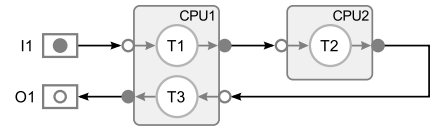
Input streams	I1: periodic (P = 60ms) I2: periodic (P = 5ms) I3: periodic (P = [60..110]ms)
Execution times	T1: 35ms, T2: 2ms, T3: 4ms, T4: 12ms
Scheduling parameters	priority T1: high, priority T2: low priority T3: low, priority T4: high

Fig. 5: Specification of Benchmark 1



Input streams	I1: periodic (P = 100ms) I2: periodic (P = 5ms)
Execution times	T1: 2ms, T2: 2ms, T3: [2..22]ms, T4: 1ms
Scheduling parameters	priority T1: high, priority T2: low priority T3: high, priority T4: low

Fig. 6: Specification of Benchmark 2



Input stream I1	periodic with burst (P = 10ms, J = [0..50]ms)
Execution times	T1: 1ms, T2: 4ms, T3: 4ms
Scheduling par.	priority T1: low, priority T3: high

Fig. 7: Specification of Benchmark 3, Scenario 2

techniques for analysis of real-time systems, and compared four analysis frameworks. MAST [9] represents the class of holistic analysis. It is based on a notion of transactions that, in case of task dependencies, are propagated through the system. The two well established tools SymTA/S [18], [17] and MPA-RTC [6] represent compositional techniques. While SymTA/S is based on fixed point calculation of event streams, MPA-RTC is based on real-time calculus. Finally, the framework of Hendriks and Verhoef [10] (in the following abbreviated as HV) performs analysis based on timed automata, where systems are translated to UPPAAL.

We consider the HV approach as litmus test for comparison as it is assumed to provide exact results. Only if the results of our implementation coincide with the results of the HV approach we claim correctness for the given set of benchmarks.

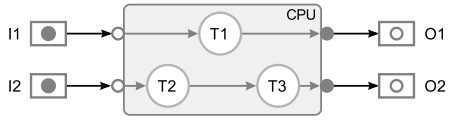
Benchmark 1: Complex Activation Pattern: This benchmark confronts the analysis frameworks with complex task activation patterns, which cannot be precisely described by *standard event models* consisting of the three parameters *period*, *jitter* and *minimum interarrival time*. Figure 5 shows the specification of Benchmark 1. Four tasks are allocated to two CPUs, each of which hosts two tasks. Note that for this and the following benchmarks all CPUs execute tasks following a preemptive fixed priority scheduling strategy. Each analysis framework is set up to determine the worst case latency of task *T3*, while gradually increasing the period of the input event stream *I3*. The activation pattern of *T3* is complex because task *T1* may preempt task *T2*.

Benchmark 2: Variable Feedback: This benchmark confronts the analysis frameworks with a feedback loop, and the resulting correlations between activations of tasks on that feedback path. Figure 6 shows the specification of Benchmark 2. Four tasks are allocated to two CPUs, each of which hosts two tasks. Each analysis framework is set up to determine the worst case latency from *I2* to *O2*, while gradually increasing the execution time of task *T3*. The input event stream *I2* is sequentially processed by three tasks, which form a feedback loop between the two CPUs.

Benchmark 3: Cyclic Dependencies: This benchmark confronts the analysis frameworks with a cyclic dependency. Figure 7 shows the specification of Benchmark 3. Three tasks are allocated to two CPUs. We only consider Scenario 2 as described in [16], as Scenario 1 does not exhibit a cyclic dependency and gives rise to the same phenomenon as Benchmark 2. Since task *T3* has higher priority than *T1*, there is a cyclic dependency between them. This is because task *T1* indirectly causes activation of *T3*, but task *T3* preempts *T1*. Each analysis framework is set up to determine the worst case latency from *I1* to *O1*, while gradually increasing the jitter of the input event stream *I1*.

Benchmark 4: Data Dependencies: This benchmark confronts the analysis frameworks with a data dependency between tasks allocated to the same CPU. Figure 8 shows the specification of Benchmark 4. Three tasks are allocated to a single CPU. Tasks *T2* and *T3* have a data dependency, as *T3* is activated by *T2*. Each analysis framework is set up to determine the worst case latency from *I2* to *O2*, while gradually increasing the execution time of task *T1*.

Evaluation Results: The evaluation of the four benchmarks showed that the analysis results of our approach perfectly match those of the HV approach. In Figure 9 the results are depicted as presented in [16]. For each benchmark the graphs show the performance characteristic depending on the parameter variance considered for the particular benchmark. Not surprisingly, the results determined by the analysis approaches based on analytical methods are over-approximations of the exact performance characteristic. This is due to the



Input streams	I1: periodic (P = 80ms) I2: periodic (P = 50ms)
Execution times	T1: [15..30]ms, T2: 20ms, T3: 10ms
Scheduling parameters	priority T1: high, priority T2: medium, priority T3: low

Fig. 8: Specification of Benchmark 4

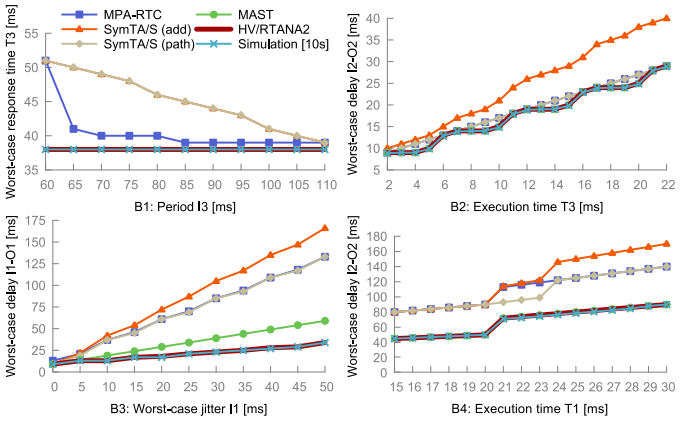


Fig. 9: Result Comparison

	B1	B2	B3	B4
HV (UPPAAL)	5...22	< 1	0.1...1	< 1
RTANA$_2$	< 1	< 1	0.1...5.5	< 1

TABLE I: Runtime Comparison (sec.)

abstraction of analytical approaches because of the inevitable limited accuracy of the underlying formalisms to capture complex dependencies between different events. On the other hand analysis approaches based on model-checking like the HV approach and the approach presented in this paper, are able to capture those event dependencies while constructing the state-space of the system. However, there is no free lunch as model-checking suffers from potential state-space explosion.

Although performance comparison is not in the focus of our evaluation, we conclude the section with some runtime results. All measurements have been performed at the same machine, thanks to the fact that all benchmark models of the HV approach are publicly available. As shown in Table I, runtimes for both approaches are below $1s$ for benchmarks 2 and 4. To our own surprise, the runtimes of the remaining benchmarks do not show significant better performance of one of the tools. We expected to see that our explicit state representation will generally impose larger runtimes than UPPAAL using a symbolic state representation, which is obviously not the case. However, as interfaces are defined by ω-languages, FSMs are the natural formalism to capture behaviour for the considered class of systems. Performance improvements by exploiting more compact state representations is a part of future work.

V. CONCLUSION

The paper revisits an existing notion of real-time interfaces and extends it with ports characterising events for task activations and completions, and by allowing us to express executions on multiple resources. These extensions enable the interface notion for more complex designs such as with task dependencies and distributed architectures, while preserving desired properties such as proper composition and refinement operations. It also paves the way for combination with contract-based design methods as it allows for a tighter integration of both formalisms.

The paper presents an initial implementation of the approach, which performs holistic response-time analysis for a particular class of real-time interfaces. Feasibility of the implementation has been shown by a set of small benchmarks.

A possible future research direction concerns more efficient state representations and operations. More important however is the extension towards compositional reasoning and refinement checks as intended by the definition of the interface formalism. Supporting compositional reasoning would also help in partly mitigate the state-space explosion problem.

REFERENCES

[1] R. Alur and G. Weiss. Regular specifications of resource requirements for embedded control software. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 159–168, 2008.

[2] R. Alur and G. Weiss. RTComposer: a framework for real-time components with scheduling interfaces. In *Proc. 8th International Conference on Embedded Software, EMSOFT*, pages 159–168, 2008.

[3] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: a Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In *Proc. FORMATS'03*, 2003.

[4] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen. Contracts for Systems Design. 2013. INRIA Research Report No. 8147 (November 2012), to appear in Proc. of the IEEE.

[5] P. Bhaduri and I. Stierand. A proposal for real-time interfaces in speeds. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 441–446, march 2010.

[6] S. Chakraborty, S. Kunzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Proc. Design, Automation and Test in Europe (DATE)*, 2003.

[7] A. David, J. I. Rasmussen, K. G. Larsen, and A. Skou. *Model-Based Design for Embedded Systems*, chapter Model-based Framework for Schedulability Analysis Using Uppaal 4. CRC Press, 2009.

[8] L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software*, EMSOFT '01, pages 148–165. Springer-Verlag, 2001.

[9] G.M. Harbour, J.J. Gutierrez Garcia, J.C. Palencia Gutierrez, and J.M. Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Euromicro Conference on Real-Time Systems*, 2001.

[10] M. Hendriks and M. Verhoef. Timed Automata based Analysis of Embedded System Architectures. In *Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[11] T.A. Henzinger and S. Matic. An Interface Algebra for Real-Time Components. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pages 253–266, 2006.

[12] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[13] G. Madl, S. Abdelwahed, and D. C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *Real-Time Syst.*, 33(1-3):77–100, July 2006.

[14] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[15] E. F. Moore. *Gedanken-experiments on Sequential Machines*, pages 129–153. Princeton University Press, 1956.

[16] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *Proc. Intl. Conference on Embedded Software*, 2007.

[17] K. Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technical University of Braunschweig, 2005.

[18] K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MpSoC Performance Verification. *IEEE Computer*, 36(4):60–67, april 2003.

[19] A. Sangiovanni-Vincentelli. Quo Vadis SLD: Reasoning about Trends and Challenges of System-Level Design. *Proceedings of the IEEE*, 95(3):467–506, March 2007.

[20] L. Sha, T. Abdelzaher, K. E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 2004.

[21] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. Funstate-an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration Systems*, 9(4):524 –544, 2001.

[22] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *Proceedings of the 6th International Conference on Embedded Software*, EMSOFT, pages 34–43, 2006.

[23] K.W. Tindell, A. Burns, and A.J. Wellings. Allocating hard real-time tasks: An NP-Hard problem made easy. *Real-Time Systems*, 1992.

[24] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *Proc. 5th International Conference on Embedded Software*, pages 80–89, 2005.

[25] E. Wandeler and L. Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *Proc. 12th Real-Time and Embedded Technology and Applications Symposium*, RTAS, pages 243–252, 2006.

[26] G. Weiss and R. Alur. Automata based interfaces for control and scheduling. In *Hybrid Systems: Computation and Control, 10th International Workshop, HSCC*, volume 4416 of *LNCS*, pages 601–613, 2007.