Virtual Integration of Real-Time Systems based on Resource Segregation Abstraction*

Ingo Stierand[1], Philipp Reinkemeier[2], and Purandar Bhaduri[3]

[1] University of Oldenburg, Germany
stierand@informatik.uni-oldenburg.de
[2] OFFIS, Germany
reinkemeier@offis.de
[3] IIT Guwahati, India
pbhaduri@iitg.ernet.in

Abstract. Embedded safety-critical systems must not only be functionally correct but must also provide timely service. It is thus important to have rigorous analysis techniques for determining timing properties of such systems. We consider a layered design process, where timing analysis applies when the system is integrated on a target platform. More precisely, we focus on contract-based design, and ask whether a set of real-time components continues to comply to a given system specification when it is integrated on a common hardware.
We present an approach for compositional timing analysis, and define conditions under which the system integration will preserve all the timing properties given by the system specification. Therefore, engineers can negotiate specifications of the individual components a priori, knowing that no integration issues will occur due to shared resource usage. The approach exploits ω-languages, which enables analysis techniques based on model-checking. Such an analysis is shown by a case study.

1 Introduction and Related Work

Developing safety-critical real-time systems is becoming increasingly complex due to the growing number of functions realized by these systems. Moreover, an increasing number of functions are realized in software, which are then integrated on a common target platform in order to save costs. The integration on a common platform causes interferences between the different software functions due to their shared resource usage. It is desirable to bound these interferences in a way to make guarantees about the timing behavior of the individual software-functions. A schedulability analysis delivers such bounds for interferences between software-tasks sharing a CPU by means of a scheduling strategy.

* This work was partly supported by the Federal Ministry for Education and Research (BMBF) under support code 01IS11035M, *Automotive, Railway and Avionics Multicore Systems* (ARAMiS), and 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).

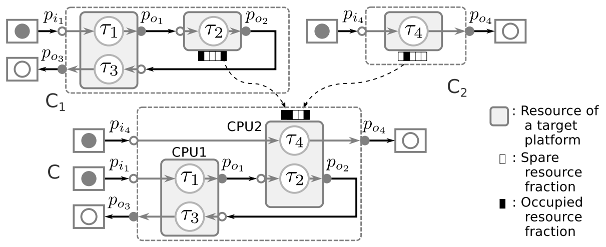


Fig. 1. Exemplary Integration Scenario using Resource Segregation

The platform integration is typically part of a larger design process with different phases. For safety-critical system design it is crucial that, starting with the initial design, all steps ensure that the final implementation indeed satisfies all given requirements. Contract-based design [2] provides a formal foundation allowing us to reason about the validity of a design in all phases. Based on well-defined semantics and operations, all design steps can be checked to verify the result still satisfies the overall system requirements.

Formal verification, such as with contracts, is however not an easy task, and requires carefully selected approaches in order to tackle computational complexity. We focus on the integration phase, where real-time components are allocated to the hardware platform. We present a compositional analysis framework using real-time interfaces based on ω-regular languages. Following the idea of interface-based design, components are described by interfaces and can be composed if their corresponding interfaces are compatible. The contribution of this work allows us to formally capture the resource demand of an interface, which we call *segregation property*. Compatibility of interfaces then can be reduced to compatibility of their segregation properties. Additionally, we put this into the context of contract-based design, enabling us to reason about the overall specification satisfied by the integrated implementation in a compositional way.

More specifically, we consider the following scenario. The bottom part of Figure 1 shows a target platform that is envisioned by say an Original Equipment Manufacturer (OEM). It consists of two processing nodes (CPU_1 and CPU_2). Suppose the OEM wants to implement two applications, components C_1 and C_2, on this architecture and delegates their actual implementation to two different suppliers. Both applications share a subset of the resources of the target platform, e.g. tasks τ_2 and τ_4 are executed on CPU_2 after integration. Furthermore, we assume the system specification C shown in Figure 1 to be given from previous design phases. While some components together with their (local) specifications may also be known (e.g. in case of reuse), the OEM generally has to negotiate proper specifications with the suppliers, in our case C_1 and C_2.

Now two tasks have to be accomplished: It must be ensured that (1) the composition of C_1 and C_2 conforms to the specification C, and (2) the composed implementation satisfies C as well. It is highly desirable that both tasks are performed before the suppliers start to implement the respective components. Later integration issues would require to repeat this step, causing increased development time and costs. To this end, the negotiation between the OEM and the suppliers must include the resource consumption needed by the implementations. Otherwise, the contract theory will fail to detect integration issues that may occur due to shared resource usage. We therefore assign a resource reservation to each component, guaranteeing a certain amount of resource supply. Then the timing behavior of both components can be analyzed independently from each other based on their resource demands and the guaranteed resource supply. Verification of the successful integration of C_1 and C_2 then amounts to checking whether the reserved resource supplies can be composed. We further define conditions under which the integrated application will satisfy its system specification. These conditions allow us to derive proper (real-time) specifications for the negotiation with the suppliers, and hence to tackle the first task.

There has been a considerable amount of study on compositional real-time scheduling frameworks [11, 12, 9, 6, 4]. These studies define interface theories for components abstracting the resource requirement of a component by means of demand functions [11, 12], bounded-delay resource models [6], or periodic resource models [9, 4]. Based on these theories the required resources of a component, captured by its interface, can, for example, be abstracted into a single task. This approach gives rise to hierarchical scheduling frameworks where interfaces propagate resource demands between different layers of the hierarchy. Our proposed resource segregation abstraction is an extension of the real-time interfaces presented in [3]. Contrary to the aforementioned approaches, our real-time interfaces and resource segregation are based on ω-regular languages. This means the approach can for example be employed in automata-based model-checking frameworks. In addition the results we present are not bound to specific task and resource models, like periodic or bounded delay.

Analytical methods provide efficient analysis by abstracting from concrete behavior. This, however, typically leads to over-approximations of the analysis results. Computational methods on the other hand, such as model-checking for automata ([1, 7, 5]), typically provide the expressive power to model and analyze real-time systems without the need for approximate analysis methods. This flexibility comes with costs. Model-checking is computationally expensive, which often prevents analysis of larger systems. The contribution of this paper will help to reduce verification complexity for the application of computational methods.

The paper is structured as follows: We start with an introduction of real-time interfaces as presented in [10], which characterize components including their resource demands. Section 3 recapitulates the basic notions of contract-based design that are consistent with our interfaces. Sections 4 and 5 provide the notions and results to reason about the integration of interfaces in a compositional way in the context of contract-based design. Section 6 shows the application of the approach by an example, and Section 7 concludes the paper.

2 Real-Time Interfaces

A real-time interface characterizes a component when it is executed on a set of resources such as processing nodes and buses. Each interface represents a set of real-time tasks, and specifies a set of legal schedules when it is executed on

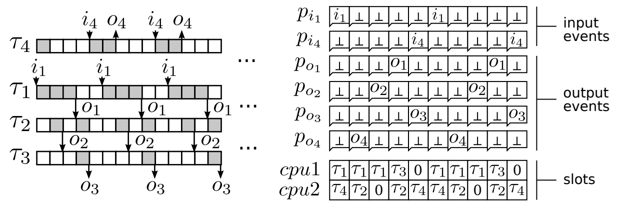


Fig. 2. Scheduling scenario (left) and exemplary trace-extract (right).

the resources in discrete slots of some fixed duration. For example, consider a component with two tasks, τ_1 and τ_3, which are scheduled on a single resource. A schedule for this component can be described by an infinite word over the alphabet $\{0, \tau_1, \tau_3\}$, where 0 means the resource is idle during the slot, and τ_1 and τ_3 means the corresponding task is running.

Example 1. Suppose that task τ_1 is a periodic task with period $p = 5$ and execution time $c = 3$. The slot language of its interface can be described by the regular expression $0^{<5}[\tau_1^3 \,|||\, 0^2]^\omega$, where $u \,|||\, v$ denotes all possible interleavings of the finite words u and v. That means, a schedule is legal for the interface, as long as it provides 3 slots during a time interval of length 5. Observe that the slot language 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 by lower and upper arrival curves $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$ [8].

As interfaces also capture task activations and completions, we consider languages over tuples of symbols. A component has a set $P = P_{in} \uplus P_{out}$ of input and output ports. Symbols occurring at a port represent activation and completion events for the tasks that are connected to this port. The events observed at port $p \in P$ are characterized by the alphabet Σ_p, and we define $\Sigma_P = \Sigma_{p_1} \times \ldots \times \Sigma_{p_n}$. As task activations and completions need not occur at each time step, we define a special symbol $\perp$ denoting that no event occurs. Interfaces talk about 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, as shown above.

Definition 1. *An* interface *is a tuple* $I_K = (K, \Sigma_K, L_K)$ *where* $K = P \cup R$ *is a set of ports* P *and resources* R, $\Sigma_K = \prod_{k \in K} \Sigma_k$, $L_K \subseteq \Sigma_K^\omega$, *and:*

- *For* $k \in P, \Sigma_k$ *is the set of events,* $\perp \in \Sigma_k$, *that may occur at port* k.
- *For* $k \in R, \Sigma_k$ *is the set of tasks,* $0 \in \Sigma_k$, *that run on resource* k. $\diamond$

Example 2. Suppose task τ_1 on the system depicted at the bottom of Figure 1 is as in Example 1. Task τ_2 depends on τ_1, i.e., is activated by τ_1, and has an execution time $c_2 = 2$. Task τ_3 depends on τ_2 and has an execution time $c_3 = 1$. Task τ_4 is also a periodic task with period $p_4 = 5$ and $c_4 = 2$. Suppose both CPUs are scheduled using a fixed priority preemptive policy, where tasks τ_1 and τ_4 have high priority on their CPU. The delay of the task-chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ depends on the activation-pattern of τ_4 and its execution time. This is illustrated on the left of Figure 2. Once τ_1 completes execution it activates (via port p_{o_1}) τ_2, which in turn might be preempted by τ_4. Finally, τ_3, activated by τ_2, 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 $I_K = (K, \Sigma_K, L_K)$, $K = P \cup R$, $P = \{p_{i_1}, p_{i_4}, p_{o_1}, \ldots, p_{o_4}\}$ and $R = \{cpu1, cpu2\}$, $\Sigma_{p_{i_j}} = \{i_j, \perp\}$, $\Sigma_{p_{o_j}} = \{o_j, \perp\}$, $\Sigma_{cpu1} = \{\tau_1, \tau_3, 0\}$ and $\Sigma_{cpu2} = \{\tau_2, \tau_4, 0\}$. An excerpt of a possible trace in L_K is shown in Figure 2, which corresponds to the discussed scheduling scenario. Observe that every port has its own event tape in the interface, just as each resource has its own tape of time slots. Note that we omitted input ports connected to some output port: we define a connection between tasks by a unification of their ports to characterize a synchronization of the behavior.

The key to dealing with interfaces having different alphabets is a *projection* operation. For alphabet Σ, language $L \subseteq \Sigma^\omega$, and $\Sigma' \subseteq \Sigma$, we consider the projection $proj(\Sigma, \Sigma')(L)$ to Σ', which is the unique extension of the function $\Sigma \to \Sigma'$ that is identity on the elements of Σ' and maps every element of $\Sigma \setminus \Sigma'$ to 0. We will also need the *inverse projection* $proj^{-1}(\Sigma'', \Sigma)(L)$, for $\Sigma'' \supseteq \Sigma$, which is the largest language over Σ'' whose words projected to Σ belong to L. We further define $proj(\Sigma, \emptyset)(L) := \emptyset$, and $proj^{-1}(\Sigma'', \emptyset)(\emptyset) := \Sigma''^\omega$.

For alphabets of the form $\Sigma_K = \Sigma_{k_1} \times \ldots \times \Sigma_{k_n}$, the projection operation is performed component-wise, i.e., for each k_i individually. Furthermore, we want to consider interfaces over different index sets. To this end, we define *normalization* 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 $\Delta_K|_{K'} = \prod_{k \in K'} \Delta_k$. We extend these operations to words and languages, i.e., we define $\omega_{K' \to K}$, $L_{K' \to K}$, $\omega_K|_{K'}$ and $L_K|_{K'}$, respectively.

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 $i \in N$. 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. The inverse projection of a word over Σ_i results in a set of words where every 0 in the word is 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.

This notion of interfaces exhibits several interesting operations and properties [10]. In the considered context the composition operation is of importance, which obtains the set of schedules when two components are executed together:

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

– $K = K_1 \cup K_2$,

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

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 the 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.

Note that the projection operation also captures the synchronization of the connected ports of I_1 and I_2, i. e., which events are synchronized in the composition. This is illustrated in Figure 2. Ports connected in the system are unified in the corresponding interface (e.g. port p_{o_1}), which means the same behavior can be observed at connected ports. In the following, we will write $L_1 \breve{\cap} L_2$ for inverse projection followed by intersection when the common target alphabet is known from the context. So we could write $L_K = L_{K_1} \breve{\cap} L_{K_2}$ in Definition 3.

3 Contracts and Virtual Integration

While our interfaces are suitable for expressing concurrent resource usage of an implementation, contracts are a suitable notion for specifications in upstream design phases. A main advantage of contract-based design is to distinguish explicitly responsibilities of the individual parts of a design. A contract is a pair (A, G) of assertions where A is an *assumption* about the environment of a component, and G is the *guarantee* the component offers to its environment [2]. Logically, this is equivalent to $A \Rightarrow G$. In the context of this paper, both assumptions and guarantees will talk about bounds on the frequency of task arrivals and time to completions. In addition, they capture 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. Again, these are modeled using ω-regular languages. The semantics of a contract is about the behavior 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 (modeled by $\perp$), a task arrives (modeled by an event at the input port of the task) or finishes execution (modeled 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 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 $A, G \subseteq \Sigma_P^\omega$.

An important objective of any design process is successive refinement. The contract theory provides the corresponding relation that states whether a specification C' refines another specification C. Indeed this is the case if C' can be used in any context as C, and if C' has a restricted behavior:

Definition 4. *[2] A contract C' refines another contract C, written $C' \preceq C$ if and only if $A \subseteq A'$ and $G' \subseteq G$.* $\diamond$

As the ultimate goal of the design process is to obtain an implementation, we also need to define under which conditions an implementation behaves as specified:

Definition 5. *[2] Let $C = (P, \Sigma_P, A, G)$ be a contract. An implementation M of the contract* satisfies *C, written $M \models C$, if and only if $M \cap A \subseteq G$.* $\diamond$

Note that contract refinement and satisfaction are consistent. When an implementation M satisfies a contract C', and C' refines C, then M also satisfies C. In our scenario, we indeed consider interfaces as implementations.

The last important operation in the present setting is contract composition. Systems are build from individual parts that are put together in order to provide the intended functionality. In a bottom-up design, the composed contract C is obtained from the contracts of the composed components. The operation is based on the observation that the assumption of a composed contract shall be the maximal behavior that does not cause integration errors. For contracts $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$, the contract $C = C_1 \parallel C_2$ is given by:

$$A = \max\{A \mid A \cap G_1 \subseteq A_2 \land A \cap G_2 \subseteq A_1\} \tag{1}$$

$$G = G_1 \cap G_2 \tag{2}$$

For ω-languages the equations above result in the following definition. Note that, in order to reason about contracts over different port sets, the alphabets of the involved assertions must be made equal. 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 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*

$$A = (A'_1 \cap A'_2) \cup (A'_1 \cap \overline{G'_1}) \cup (A'_2 \cap \overline{G'_2}), \qquad G = G'_1 \cap G'_2,$$

where $A'_i = proj^{-1}(\Sigma_P, \Sigma_{P_i})(A_i)$, $G'_i = proj^{-1}(\Sigma_P, \Sigma_{P_i})(G_i)$. $\diamond$

In a top-down design process we assume the system specification to be given. Though some components might be known (e.g. from previous versions of the design), the designers have a good understanding of what the system shall do. In this case, one can derive from Eq. (1) and (2) the conditions under which a system composed of individual parts conforms to a given specification. We call them *virtual integration conditions*:

Lemma 1. *For contracts $C = (A, G)$, $C_1 = (A_1, G_1)$ and $C_2 = (A_2, G_2)$ the following holds: $C_1 \parallel C_2 \preceq C$ if and only if $A \cap G_1 \subseteq A_2$ and $A \cap G_2 \subseteq A_1$ and $G_1 \cap G_2 \subseteq G$.* $\square$

4 Compositional Virtual Integration

One important property of contract based design is that contracts can be *independently implemented.* In [2] this is formalized as follows: For all contracts

C_1, C_2, C_1' and C_2', if $C_1' \preceq C_1$ and $C_2' \preceq C_2$ hold, then $C_1' \parallel C_2' \preceq C_1 \parallel C_2$. Thus, contracts can be *independently* refined towards a final implementation and composing these implementations always results in an implementation of the composed contracts. So considering real-time interfaces as implementations, one might expect: Given a system specification C, it can be decomposed into contracts C_i negotiated with suppliers. If $C_1 \parallel ... \parallel C_n \preceq C$ holds, as well as $I_i \models C_i$, then $I_1 \parallel ... \parallel I_n \models C$. However, it can happen that reasoning about integration based on the introduced contract formalism fails to detect integration issues when composing interfaces. To give an example, consider contracts C_1 and C_2, each specifying an assumption about events occurring with a period interval $[5, 6]$ at an input port, and as guarantee a deadline of 6 between events occurring at that input port until an event is sent at an output port. Now assume interfaces $I_1 \models C_1$ and $I_2 \models C_2$, each of which exactly mirrors the input and output behavior of C_1, C_2 respectively. Each interface has a single task with an execution time of 3 and both tasks share the same resource. Observe that all the formulas from above hold. However, $I_1 \parallel I_2$ only accepts input behavior for both input ports with a strict period of 6. This is due to the incompatible resource usages, i.e. the tasks are not schedulable under the assumed activation rates. Of course this is not what we want, since the assumption of $C_1 \parallel C_2$ tells us that a valid environment may send events with a period interval of $[5, 6]$. Hence, relying on the assumption and using $I_1 \parallel I_2$ in a context, where the environment sends events with a period of 5 would cause deadlines of C_1 and C_2 to be missed.

The cause of this problem is twofold: First, our specification in terms of contracts is incomplete. Since the contracts do not talk about resource usage, there is simply no way to detect integration errors due to resource sharing solely based on them. Second, satisfaction as per Definition 5 does not force the implementation to accept every behavior expressed by the assumption.

Our solution to these problems is, first, to define a notion of *characteristic contract* of an interface. This allows us to use the stricter contract refinement relation instead of satisfaction. Second, we define a *composability* criterion for interfaces, which avoids integration errors when composing them. As we consider real-time interfaces as implementations, we develop sufficient conditions for interface composability, which can be checked based on contracts and an abstraction of the resource usage of the interfaces. The latter allows us to check for proper integration in design phases before the actual implementation exists.

For the characteristic contract of an interface, we focus on the case where the assumptions define activations patterns for each input port, and the guarantees define execution deadlines [10]. As an interface includes the behavior observed at the component ports, it can serve as a specification. Expressing such a combined specification of assumption and guarantee as a contract is in general not easy. However, if one is interested in assumptions that talk only about the behavior of the input ports, as in our case, it becomes straightforward:

Definition 7. *Let I be an interface, and P be the set of ports in the index set $K = P \cup R$ of I. We define the* characteristic contract $C_I = (A_I, G_I)$ *of I, where $A_I = L_I|_{P_{in}}$ and $G_I = L_I|_P$.* ◇

As we have observed, composition of interfaces may restrict their accepted input behavior. The goal is to define *composability* of interfaces, such that this restriction does not occur. More formally, for composable interfaces the following should hold: $C_{I_1\|\ldots\|I_n} \preceq C_{I_1} \parallel \ldots \parallel C_{I_n}$. For the definition of composability, we need a notion of "maximal resource usage" that allows us to reason about the maximum resource demand of an interface:

Definition 8. *Let $\omega = \sigma_0\sigma_1\ldots$ and $\omega' = \sigma'_0\sigma'_1\ldots \in \Sigma^\omega$ where $0 \in \Sigma$. We say $\omega' \le \omega$ if and only if $\forall i \in \mathbb{N} : \sigma_i = \sigma \implies \sigma'_i \in \{0, \sigma\}$. We extend this to words over tuple of symbols: Let be $\omega_K, \omega'_K \in \Sigma^\omega_K$. We say $\omega'_K \le \omega_K$ if and only if $\forall k \in K : \omega'_K|_k \le \omega_K|_k$.* $\diamond$

A word ω' precedes ω if either both words agree on the usage of each slot σ_i, or that slot is not used in ω' (i.e. $\sigma'_i = 0$). In other words, a slot used in ω' ($\sigma'_i \neq 0$) is also used in ω. We extend this order on slot words to languages over Σ^ω_K:

Definition 9. *Let be $L_K, L'_K \subseteq \Sigma^\omega_K$. We define $L'_K \sqsubseteq L_K$ if and only if $\forall \omega'_K \in L'_K : \exists \omega_K \in L_K : \omega'_K \le \omega_K$.* $\diamond$

Intuitively, $L'_K \sqsubseteq L_K$ means that the slot usage of *all* words $\omega'_K \in L'_K$ is "dominated" by *at least one* word $\omega_K \in L_K$. Note that $(\mathcal{P}(\Sigma^\omega_K), \sqsubseteq)$ is a preorder, as $L_K \sqsubseteq L'_K$ and $L'_K \sqsubseteq L_K$ does not necessarily imply $L_K = L'_K$. We are interested in a particular subset of a slot language $L_K \subseteq \Sigma^\omega_K$, containing only those words from L_K with maximal execution demands:

Definition 10. *Given a slot language $L_K \subseteq \Sigma^\omega_K$, we define $\widehat{L_K} = \{\omega_K \in L_K \mid \forall \omega'_K \in L_K : \omega_K \le \omega'_K \Rightarrow \omega_K = \omega'_K\}$* $\diamond$

Intuitively, $\widehat{L_K}$ removes all words from L_K, whose slot usage is "dominated" by another word in L_K. $\widehat{L_K}$ is unique and maximal with respect to the order $\sqsubseteq$:

Lemma 2. *For every $L_K \subseteq \Sigma^\omega_K$ the subset $\widehat{L_K} \subseteq L_K$ is unique and maximal, i.e. $\forall L'_K \subseteq L_K : L'_K \sqsubseteq \widehat{L_K}$.* $\square$

Now we can define the conditions for composability of interfaces, based on Definition 10 and Lemma 1:

Definition 11 (Composability of Interfaces). *Let I_1 and I_2 be two interfaces. We say I_1 and I_2 are* composable *if:*

1. $L_1|_{P_{1in}} = \prod_{p \in P_{1in}} L_1|_p$ *and* $L_2|_{P_{2in}} = \prod_{p \in P_{2in}} L_2|_p$
2. $L_1|_{P_{2in}} \subseteq A_{I_2}$ *and* $L_2|_{P_{1in}} \subseteq A_{I_1}$
3. $\forall a \in L_1|_{P_{1in}} \breve{\cap} L_2|_{P_{2in}} : \widehat{L_1(a)|_{R_1}} \breve{\cap} \widehat{L_2(a)|_{R_2}} \neq \emptyset$, *where* $L_j(a) = \{\omega \in L_j \mid \omega|_{P_{jin}} = a|_{P_{jin}}\}$. $\diamond$

The first condition requires the behavior specified for the individual input ports of every component to be independent. The second one provides the virtual integration condition as in Lemma 1 (note that $L_i = A_{I_i} \cap G_{I_i}$). And the third condition requires that all components can be executed even if they expose maximal execution usage, considered separately for every possible activation.

The following result states that under these conditions the involved interfaces can indeed be composed without restricting their original input specification:

Theorem 1. *Let I_1 and I_2 be composable interfaces, and let $I = I_1 \parallel I_2$. Then $A_I = (A_{I_1} \breve{\cap} A_{I_2})|_{P_{in}}$.* □

This result establishes the requested properties: Considering interfaces as implementations of their characteristic contracts, we can check whether their composition restricts input behavior:

Corollary 1. *Let I_1 and I_2 be composable interfaces. Then the following holds: $C_{I_1 \parallel I_2} \preceq C_{I_1} \parallel C_{I_2}$.* □

We use the above results to solve our initial integration problem as follows. Given the system specification $C = (A, G)$, the OEM can decompose it into sub-contracts $C_1 ... C_n$ during the negotiation phase with the suppliers. The property $C_1 \parallel ... \parallel C_n \preceq C$ establishes Condition 2) of Definition 11, provided the characteristic contracts of the interfaces implemented by suppliers refine their local sub-contract, i.e. $C_{I_i} \preceq C_i$. This is the responsibility of the suppliers. Further the assumption of each C_{I_i} must be such that Condition 1) of Definition 11 is satisfied. Condition 3) requires the interfaces to be known, which still remains an obstacle in the design flow. In the remaining part we introduce *segregation properties*, providing sufficient conditions to establish the third condition of Definition 11, which can be negotiated without knowing the final implementation.

5 Resource Segregation

A *segregation property* B_I for an interface I abstracts from the slot allocations of the legal schedules of I, by means of a set of *input-independent* slot reservations for which the interface is schedulable. Note that B_I indeed may reserve more slots than are used by the interface I. The basic idea is that composition of segregation properties B_{I_1} and B_{I_2} of interfaces I_1 and I_2 then combines non-conflicting slot reservations of B_{I_1} and B_{I_2}. If at least one such combination exists, then the third condition for composability of I_1 and I_2 holds.

Definition 12. *Let I_K be an interface, and let $B \subseteq \Sigma_R^\omega$ be a slot reservation language over R, the set of resources in K. B is a* segregation property *for I_K if and only if*

$$\forall b \in B, \forall a \in L_K|_{P_{in}} : \exists \omega \in \widehat{L_K(a)|_R} : \omega \leq b$$

where $L(a) = \{\omega \in L \mid \omega|_{P_{in}} = a|_{P_{in}}\}$ ◇

Hence B is a segregation property for I_K, if for *all* its possible activation patterns *each* word in B "dominates" at least one of the maximal execution demands resulting from the activation pattern.

The composition of slot reservation languages B_1 and B_2 is defined by $B_1 \breve{\cap} B_2$. The condition for composability of slot reservation languages is rather simple:

Definition 13. *Two slot reservation languages $B_1, B_2 \subseteq \Sigma_R^\omega$ are composable if and only if $B_1 \breve{\cap} B_2 \neq \emptyset$.* ◇

The following proposition states the desired sufficient condition for the third condition of interface composability based on their segregation properties.

Theorem 2. *Let I_1 and I_2 be interfaces with disjoint input port sets $P_{1_{in}}, P_{2_{in}}$ and disjoint output port sets $P_{1_{out}}$ and $P_{2_{out}}$. Let $B_i \subseteq \Sigma_R^\omega$ be segregation property for I_i. Then the following holds:*

$$B_1 \breve{\cap} B_2 \neq \emptyset \implies \forall a \in L_1|_{P_{1in}} \breve{\cap} L_2|_{P_{2in}} : \widehat{L_1(a)|_{R_1}} \breve{\cap} \widehat{L_2(a)|_{R_2}} \neq \emptyset$$

where $L_j(a) = \{\omega \in L_j \mid \omega|_{P_{jin}} = a|_{P_{jin}}\}$. □

From Definition 12 follows that, given a segregation property B_I, *any* non-empty subset $B'_I \subseteq B_I$ is also a segregation property for interface I. In particular every $b \in B_I$ is a segregation property for I. Further, if I is schedulable under b, then it is schedulable under any slot reservation b' with $b \leq b'$. In other words, we can always reserve more slots for I without affecting its schedulability. This leads us to the following definition for refinement of slot reservation languages:

Definition 14. *Given slot reservation languages $B, B' \subseteq \Sigma_R^\omega$, we say B' refines B (B abstracts B'), denoted $B' \preceq B$, if and only if $\forall b \in B : \exists b' \in B' : b' \leq b$.* ⋄

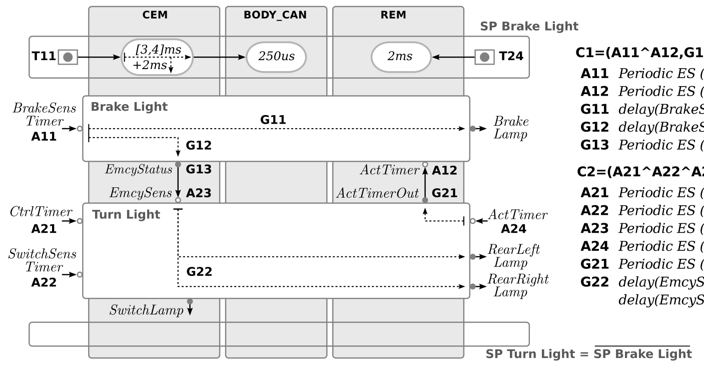
Similar to the order on slot languages, refinement for slot reservations is a pre-order $(\mathcal{P}(\Sigma_R^\omega), \preceq)$. Given a segregation property B_I, every B'_I with $B_I \preceq B'_I$ is also a segregation property for I. Hence, composability of segregation properties can be checked based on their abstractions, as stated by the following Lemma, which follows directly from Definition 14:

Lemma 3. *Let $B_1, B_2, B'_1, B'_2, \subseteq \Sigma_R^\omega$ be slot reservation languages such that $B'_1 \breve{\cap} B'_2 \neq \emptyset$, then $B_1 \preceq B'_1$ and $B_2 \preceq B'_2$ implies $B_1 \breve{\cap} B_2 \neq \emptyset$.* □

This allows us to augment contract-based design with resource reservations by associating with contract C a slot reservation B. In this combined specification (C, B), contracts are refined together with their slot reservation. Composition of (C_1, B_1) and (C_2, B_2) amounts to composing the contracts as well as their slot reservations, i.e. $(C_1, B_1) \parallel (C_2, B_2) = (C_1 \parallel C_2, B_1 \parallel B_2)$. An implementation I satisfies (C, B) if $C_I \preceq C$ and B is a segregation property for I.

6 Case Study

We apply the approach to a case study from the automotive domain as depicted in Figure 3. The system under investigation has two components that control the signal lights of a car according to the drivers actions. The **Brake Light** component controls the rear brake lights according to the drivers brake pedal position. The pedal position is periodically sensed, which is characterized by the input port *BrakeSensTimer*. The activity of the brake lights is controlled by the output port *BrakeLamp*. The **Turn Light** component controls the turn lights according to the position of the turn switch and the warning light switch at the driver console. To this end, the component senses periodically the position



C1=(A11^A12,G11^G12^G13)

A11	*Periodic ES (P,J) = (20ms,0)*
A12	*Periodic ES (P,J) = (10ms,0)*
G11	*delay(BrakeSensTimer,BrakeLamp) <= 25ms*
G12	*delay(BrakeSensTimer,EmcyStatus) <= 10ms*
G13	*Periodic ES (P,J) = (20ms,2ms)*

C2=(A21^A22^A23^A24,G21^G22)

A21	*Periodic ES (P,J) = (20ms,0)*
A22	*Periodic ES (P,J) = (20ms,0)*
A23	*Periodic ES (P,J) = (20ms,2ms)*
A24	*Periodic ES (P,J) = (10ms,0)*
G21	*Periodic ES (P,J) = (20ms,0)*
G22	*delay(EmcySens,RearLeftLamp) <= 50ms*
	delay(EmcySens,RearRightLamp) <= 50ms

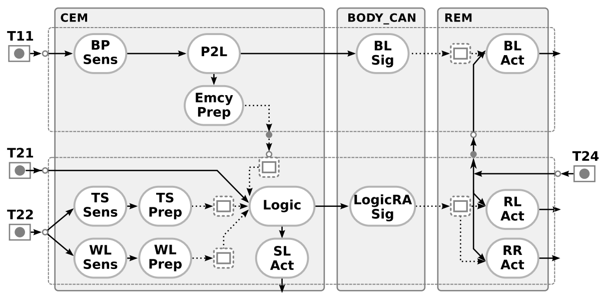
Fig. 3. Case Study: Components, Contracts and Segregation Properties

of the switches (input port *SwitchSensTimer*), and actuates the turn indicator lights accordingly. In our case, these are the lights connected to the output ports *RearLeftLamp* and *RearRightLamp*, respectively (the front lights are omitted).

The system also implements an emergency brake signaling feature. Whenever the driver performs an emergency brake (which is indicated by a brute force brake action), then the car should activate both rear turn lights in order to signal following drivers about the emergency brake situation. The emergency brake detection takes place in the **Brake Light** component. It informs the **Turn Light** component via the *EmcyStatus*/*EmcySens* port connection about the current emergency brake status, which actuates the turn lights accordingly.

The OEM defines two timing requirements for the system. The first one states that the delay between the brake sensing and the activation of the brake light must be no greater than $25ms$. The second requirement states that the end-to-end latency between the brake sensing and the activation of the warning turn lights in case of an emergency brake situation must not exceed $60ms$.

Negotiation Phase: The OEM mandates different suppliers for the two component implementations. According to our approach, the OEM specifies contracts for the individual components, which are shown at the right part of Figure 3. The assumptions **A11** and **A12** define the activation patterns for the component **Brake Light** in terms of periodic event streams as discussed in Section 2. Assumption **A11** defines a periodic event stream with a period of $20ms$ for the sensing part, and **A12** defines a periodic activation with $10ms$ period for the actuator part. A similar situation holds for the assumptions **A21** and **A22** ($20ms$), and **A24** ($10ms$) for the component **Turn Light**. Note that these assumptions anticipate an implementation detail, as the OEM defined asynchronous activations for the individual parts of the components (sensing (**A11**, **A22**), control (**A21**) and actuator (**A12**, **A24**)). The deadline requirements are expressed by contract guarantees. The first one is local to **Brake Light**, and expressed by the guarantee **G11**. The second requirement expresses an end-to-end deadline across the two components. Here, the OEM has to split the deadline into two parts. In our case, the OEM selects $10ms$ for component **Brake Light**, expressed by the guarantee **G12**, and $50ms$ for component **Turn Light** (**G22**).



Task	Execution Time (ms)
BPSens	1
P2L	2
EmcyPrep	2
BLSig	0.25
BLAct	2
Logic	4
TSSens	2
TSPrep	1
WLSens	1
WLPrep	1.5
SLAct	2
LogicRASig	0.25
RLAct	2
RRAct	2

Fig. 4. Case Study: Implementation Model

The applications shall be deployed on a hardware architecture with two ECUs. The sensing and control part shall be implemented on the ECU **CEM**, and the actuator part on the ECU **REM**. The ECUs are connected to the body CAN bus for exchanging data between the parts. The OEM provides for each part a time budget. The slot reservation for the **Brake Light** component is indicated by a set of real-time tasks (upper part of Figure 3). For example, the OEM ensures that the ECU provides $3 - 4ms$ execution time for sensing and control of the brake light part, and additional $2ms$ for the calculation of the emergency brake situation. This is expressed by the task at the top level corner of Figure 3. For the analysis, the slot reservation is represented by a finite state machine (FSM) that is generated in a separate analysis step. As **Brake Light** shall have higher priority, the slot reservation for the **Turn Light** component is simply the inverted FSM where non-occupied slots can be used, and vice versa.

Implementation: The suppliers eventually generate the implementation depicted in Figure 4. The **Turn Light** component implements sensing and pre-processing tasks for the turn switch (**TSSens**, **TSPrep**) and the warning lights (**WLSens**, **WLPrep**). The pre-processed data is read by the central **Logic** task, which generates control values for the individual actuator tasks. The data is sent via the CAN bus to the actuator tasks **RLAct** and **RRAct**, which are hosted at the ECU **REM**. The task also generates control data for the switch lights that reside in the driver console (**SLAct**). The brake pedal is sensed and preprocessed by the tasks **BPSens** and **P2L**, respectively. The latter sends the generated control values via the bus to the actuator task **BLAct**. The task **EmcyPrep** calculates whether an emergency brake took place. The dotted elements with a rectangle symbol indicate shared variables that store control values. The variables at the ECU **CEM** store the data from the pre-processing tasks. Whenever the **Logic** task is activated, it reads the stored values to generate the actuator control values. The shared variables at the ECU **REM** store these values for the actuator tasks. As the analysis has no functional (i.e. data dependency) aspect, we omit the variables in the analysis model in order to reduce computational complexity.

The trigger elements in the model (rectangles with a filled circle symbol) characterize the event streams conforming to the respective assumptions. Trigger **T11** for example implements an event stream that activates task **BPSens** every $20ms$ according to assumption **A11**.

Guarantee	**Brake Light**	**Turn Light**	integrated
G11	$[5.25, 16.25]$	-	$[5.25, 15.25]$
G12	$[5, 6]$	-	-
G22	-	$[10.25, 46.25]$	-
G12+G22	-	-	$[15.25, 45.25]$
FSM states	28.021	225.945.919	41.806.561
explored states	109.262	932.377.509	265.473.150

Table 1. Guarantee Verification Results

Analysis: We first check the proof obligations imposed by the conditions of Definition 11. The independence of the individual input ports is trivially satisfied due to the definition of independent event streams for all inports. The second condition concerns the ports that connect the involved components. In our case, the required language inclusion is given by definition, which can be checked by comparing the definitions of **A23** and **G13**, and **A12** and **G21**, respectively. In general, however, a formal language inclusion check must be performed. For the third condition, requiring the composability of the component interface, we exploit Theorem 2. Hence, it must be ensured that the composition of the individual slot reservation languages does not impose an empty language. Also this is given by definition in our case, as the slot reservation of the one component is the inverted slot reservation of the other one.

The remaining proof obligation is to check whether the implementations of the individual components satisfy the given requirements, while using only slots that are given by the respective slot reservation language. The analysis employs an evolved version of the tool RTANA$_2$ for computational real-time scheduling analysis [10]. Three analysis runs have been performed. The first analysis checks whether the **Brake Light** implementation satisfies the guarantees **G11** and **G12**. To this end, the analysis model contains both the application tasks and the slot reservation FSM. The tool performs a scheduling analysis where the application tasks can only use time slots that are available according to the slot reservation scheme. The second run does the same with the **Turn Light** implementation and guarantee **G22**, using however the inverted slot reservation of the first run. The third analysis checked the integrated model for comparison, and also performed an end-to-end analysis of the guarantees **G12+G22**. The results are shown in Table 1. Note that the number of states are consistently larger for the analysis of the separate **Turn Light** component compared to the integrated analysis. This is due to the fact that the slot reservation scheme introduces an additional non-determinism for the analysis of the component.

Verification for the guarantees **G13** and **G21** is not shown. The latter is trivially given as the events of trigger **T24** are directly put through to component **Brake Light**. **G13** can be derived from **A11** and **G12**. A formal verification would require a language inclusion check, which is however not yet available in the tool. Finally, the (over-approximated) end-to-end latency **G12+G22** can be easily derived by adding the results from the separated analysis runs.

7 Conclusion

In this paper we have proposed a compositional method to verify proper component integration at an early design stage, while taking into account resource usage of implementations of respective components. The method combines contract-based reasoning for verifying refinement of a system specification by a set of component specifications, with resource segregation properties. We provided a set of conditions for the composability of resource segregation that guarantees preservation of the validity of the contract-based refinement check, when resource usage of implementations of the contracts are considered in a later design step.

We showed the application of the approach by a case-study from the automotive domain, containing all steps of the proposed design process. The verification steps employed an extended version of the prototype analysis tool for interfaces discussed in [10] with preliminary support of segregation properties.

References

1. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Components in BIP. In: Proc. Software Engineering and Formal Methods (SEFM) (2006)
2. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.: Contracts for Systems Design (2013), INRIA Research Report No. 8147 (November 2012)
3. Bhaduri, P., Stierand, I.: A Proposal for Real-Time Interfaces in SPEEDS. In: Proc. Design, Automation Test in Europe (DATE) (2010)
4. Easwaran, A., Anand, M., Lee, I.: Compositional Analysis Framework using EDP Resource Models. In: Proc. Real-Time Systems Symposium (RTSS) (2007)
5. Guan, N., Ekberg, P., Stigge, M., Yi, W.: Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. In: Proc. Real-Time Systems Symposium (RTSS) (2011)
6. Henzinger, T., Matic, S.: An Interface Algebra for Real-Time Components. In: Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 253–266 (2006)
7. Perathoner, S., Lampka, K., Thiele, L.: Composing Heterogeneous Components for System-wide Performance Analysis. In: Design, Automation Test in Europe Conference Exhibition (DATE) (2011)
8. Richter, K.: Compositional Scheduling Analysis Using Standard Event Models. Ph.D. thesis, TU Braunschweig, Germany (2005)
9. Shin, I., Lee, I.: Periodic Resource Model for Compositional Real-Time Guarantees. In: Proc. International Real-Time Systems Symposium (RTSS). pp. 2–13 (2003)
10. Stierand, I., Reinkemeier, P., Gezgin, T., Bhaduri, P.: Real-Time Scheduling Interfaces and Contracts for the Design of Distributed Embedded Systems. In: Proc. Symposium on Industrial Embedded Systems (SIES) (2013)
11. Thiele, L., Wandeler, E., Stoimenov, N.: Real-Time Interfaces for Composing Real-Time Systems. In: Proc. International Conference on Embedded Software (EMSOFT). pp. 34–43 (2006)
12. Wandeler, E., Thiele, L.: Interface-Based Design of Real-Time Systems with Hierarchical Scheduling. In: Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 243–252 (2006)