Seamless Segregation for Multi-Core Systems*

Philipp Reinkemeier, Philipp Ittershagen, Philipp A. Hartmann, Stefan Henkler, Kim Grüttner

OFFIS – Institute for Information Technology
Oldenburg, Germany

Ingo Stierand

Carl von Ossietzky University of Oldenburg

August 5, 2013

*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).

Seamless Segregation for Multi-Core Systems

Contents

1 Introduction

Developing safety-critical real-time systems is becoming increasingly complex as the number of functions realized by these systems grows. Moreover, an increasing number of functions is realized in software, which are then integrated on the same target platform in order to save costs. Hence we observe a paradigm shift from rather loosely coupled federated architectures, where each subsystem is deployed on a dedicated hardware node, to integrated architectures with different subsystems sharing hardware nodes. This leads to a decrease in the number of hardware nodes in an architecture, and such systems are inherently more complex and difficult to design for a couple of reasons: In federated architectures the different subsystems are independent to a large degree (only the communication infrastructure between hardware nodes is shared). Thus, realizing and verifying a segregation of different functions in the time and space domain is relatively easy for such architectures, which is a prerequisite for enabling certification or qualification according to respective safety standards (e.g. ISO26262). In contrast, in integrated architectures with multiple subsystems sharing hardware nodes, this independence is compromised. On the one hand, this leads to safety related implications as the potential of error propagation from one subsystem to another subsystem increases. On the other hand, the compromised independence increases complexity of verification and validation activities, as subsystems now influence each other in a potentially undesired way due to their resource sharing. Recently, with multi-core platforms entering the embedded market, the design and verification of such systems becomes even more challenging. This is due to an increased interference between functions resulting from more shared resources like e.g. on-chip-buses and memory shared by the different cores and hence the functions executing on these cores.

For such highly integrated systems, design approaches are needed that cater for the segregation of functions or sets of functions. Bounding the allowed interferences of a set of functions induced by deploying other functions on the same hardware nodes, allows to define compositional verification strategies, which enable an efficient analysis of these complex systems. The timing behavior of a set of functions can then be analyzed based on the interference bounds without having to consider other functions deployed on the same hardware nodes. Finally, a segregation (in time) of the resources provided by the hardware architecture must be designed, such that the bound in interference considered during the verification is guaranteed.

Instead of characterizing the allowed amount of interference when sharing resources between different functions, one can also characterize the resource availability required by a function (or a set of functions) as both notions are dual. This idea is illustrated in Figure 1.1. I_1 and I_2 represent implementations of two independent functions each consisting of a set of tasks. R_1-R_4 denote the resources used by the tasks of the imple-

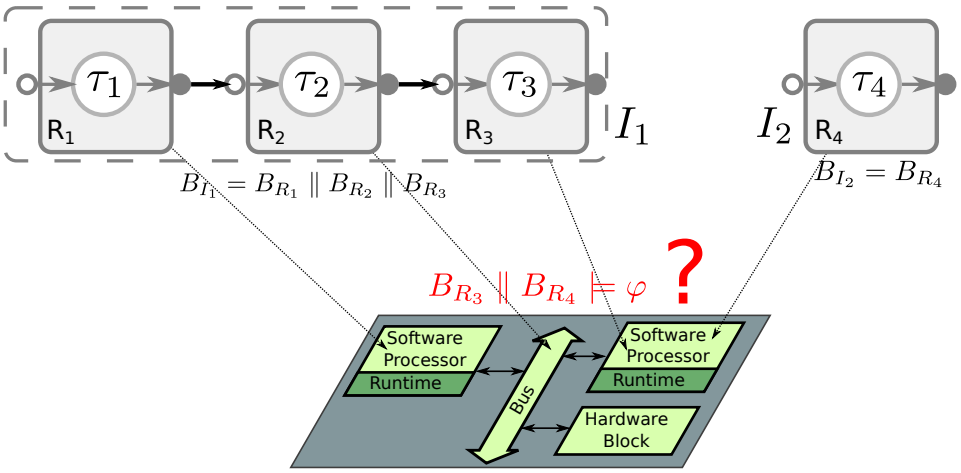


Figure 1.1: Characterizing the resource demands of implementations

mentations and B_{R_1}-B_{R_4} are characterizations of the required availability of the resources. Timing properties of I_1 and I_2 (like satisfaction of required deadlines) can be analyzed in a compositional manner based on these characterizations. If resources shall be mapped to the same components of a hardware architecture, like R_3 and R_4, characterizations of the resource availability required by the different implementations must be compatible (B_{R_3} and B_{R_4} in Figure 1.1). The compatibility of the characterizations implies that it is possible to integrate both implementations on the same hardware architecture while preserving their previously analyzed timing properties. In other words: A strategy exists for scheduling access to the resource, such that it is available to the different implementations at the times as characterized by e.g. B_{R_3} and B_{R_4}.

In this work we will present a formalism to capture the required resource availability for an implementation and an approach to timing verification employing these characterizations. The formalism is an extension of previous work pusblished in [30, 22]. Further, we will show how this formalism can be exploited in design processes based on the framework developed during the course of the SPES2020 project [9].

The SPES2020 framework proposes a meta-model for architectures of embedded systems and provides syntax and formally defined semantics of its artifacts and requirements. Using the meta-model, architecture models and requirements of embedded systems can be captured, and its semantics provide means to reason about such architecture models in a compositional way based on the requirements. A key concept of the framework is to structure models establishing different views of the developed system according to a two-dimensional schema like shown in Figure 1.2. This schema identifies a class of viewpoints, called perspectives. The perspectives are viewpoints typically found in design processes of complex embedded systems like in the avionics or automotive domain. Perspectives are complemented by abstraction levels as often systems are modeled with an increasing level of detail and refined towards an implementation.

The applicability of the meta-model when designing multi-core systems is currently elaborated in the ARAMiS project. For single-core systems, it is often sufficient to model

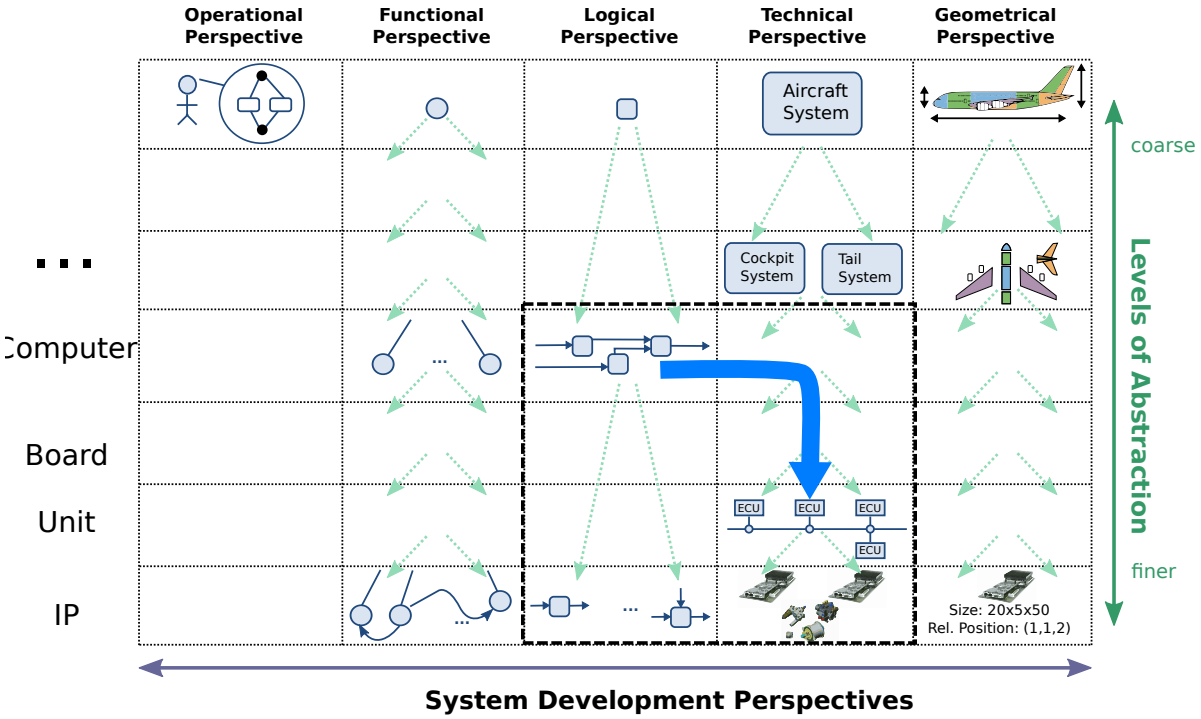


Figure 1.2: Two-dimensional design space (RTP matrix)

electronic control units as black boxes. In contrast, when designing multi-core systems, modeling control units as black boxes would abstract from details like the interconnect between the different cores and their connection to peripherals. However, these details have a great impact on the timing behavior of functions deployed on the control unit and must therefore be modeled explicitly. We believe that the models/formalisms we introduce can be used to extend the meta-model by dedicated concepts for modeling systems at these level of detail. The dashed box in Figure 1.2 highlights the area of the matrix of perspectives and abstraction levels where these kind of models would be located. Note that we consider two perspectives here, the logical perspective and the technical perspective. The deployment of functions on a multi-core architecture corresponds to a transition from a logical view of a system to a technical view (the blue arrow in Figure 1.2). During this transition it must be ensured that all requirements that are stated in the initial design of the logical perspective are satisfied consistently also by the technical realization. In our case, we focus on timing requirements such as the maximal communication delays Δ_{13} and Δ_{24} at the top left part of Figure 1.3.

To this end we introduce a notion of resource segregation (B_1-B_4 in Figure 1.3), allowing us to formally capture (sets of) time-partitions for the usage of a resource. Applying resource segregation allows analyzing deployment of different application models on the same virtual target architecture in a compositional manner. That means timing requirements of an application A_1 (e.g the Δ_{1x} in Figure 1.3), which have been proven to be satisfied under the considered set of time-partitions of the resources of the target architecture, will remain satisfied when integrated with another application A_2, if: (1) The

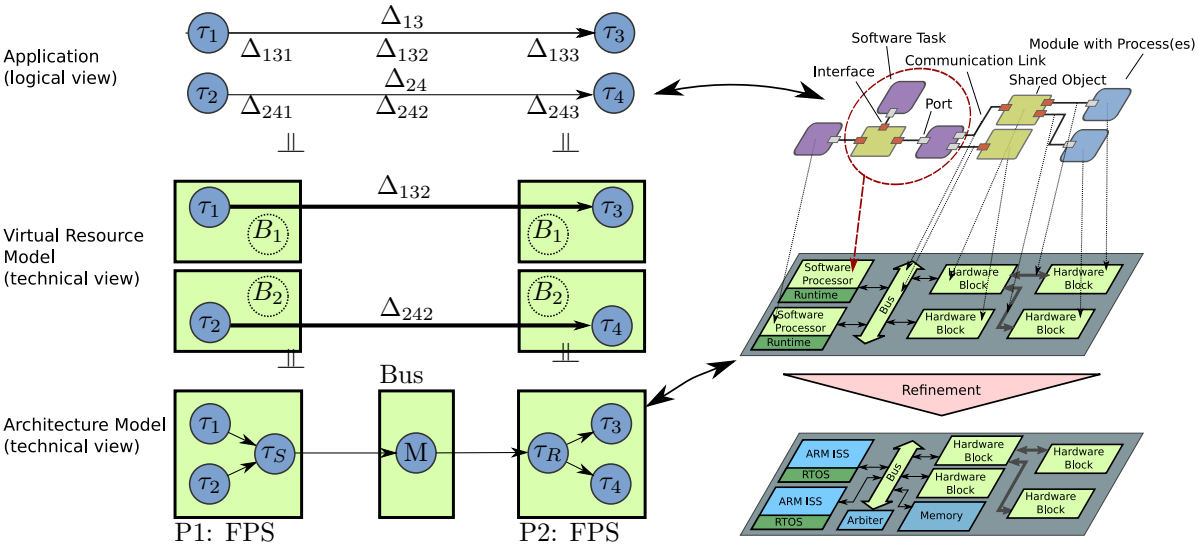


Figure 1.3: Implementing resource segregation with the OSSS simulation and refinement flow.

considered resource segregations of A_1 and A_2 have at least one time-partition that can be combined, and (2) this combined time partition is implemented on the target platform. This notion of resource segregation is an explication of the already mentioned allowed interferences (or dually the required resource availability) of an application by any other application deployed on the same target architecture. Hence, also safety-related activities in a development process can benefit from it, as an argument about the boundedness of interferences (in the time domain) between different functions now reduces to proving that the identified combined time-partition is implemented reliably on the target platform.

As discussed, to enable verfication of multicore systems, we need a precise model of the shared components, in contrast to single core verifcation approaches. As the concrete behavior of such components typically can not be completely defined in formal abstract models (due to state-space explosion), simulation based approaches are required. We propose an approach, which enables a seamless consideration of segragation from the presented logical models down to these hardware models. To handle the complexity on the hardware simulation level, a two step approach with support for virtual platform simulation is required. To enable this, we propose the integration of the formal models with the OSSS[1] methodology. OSSS is based on the SystemC language and provides concepts to model components and their behavior in terms of specialized SystemC (and therefore C++) constructs. Additionally, a rich set of architecture building blocks are provided, where modeled components can be allocated to. The simulation capabilities of SystemC allow to explore the effects of resource segregations. Finally, an implementation can be created by synthesizing configurations for supported hardware platforms as well as cross-compiling the software parts for the chosen target processors. Due to these implementation synthesis capabilities, we will focus our attention to identifying OSSS

[1]Oldenburg System Synthesis Subset

modeling patterns for multi-core systems, that support certain classes of segregation properties of the resources of a target platform. Thus, in the resulting design flow the segregation of resources is addressed seamlessly from initial application models and abstract virtual resource models down to the final implementation on a multi-core platform.

Outline

The remainder of this document is structured as follows. First, we will give a brief introduction to OSSS, the different abstraction layers and its concepts for refining a C++-based "Golden Model" on the application layer towards an implementation on a target platform. Using a platform-independent implementation of the communication protocol described in Section 2.1.3, different mapping decisions of application entities on platform components can be evaluated without changing the modeled behavior inside the tasks. In Section 2.2, we will introduce the analysis model for a formal analysis of Task Networks, based on real-time interfaces. The resource demands of these interfaces can be characterized by segregation properties.

Section 3.1 describes the mapping of OSSS modeling promitives to the introduced real-time models. Both OSSS abstraction layers will be revisited and a mapping will be presented. Finally, in Section 3.3 we present the approach how the effects of resource segregation are made transparent during the simulation os OSSS models.

2 Foundations

2.1 The OSSS Methodology

OSSS defines separate layers of abstraction for improving refinement support during the design process. The design entry point in OSSS is called the *Application Layer*. By manually applying a mapping of the system's components, the design can then be refined from the *Application Layer* to the *Virtual Target Architecture Layer*, which can be synthesized to a specified target platform in a separate step by the synthesis tool *Fossy* [11].

2.1.1 Application Layer

During the development process the separation of application and architecture is a useful concept, which enables fast design-space exploration and evaluation of system characteristics [25, 24]. The application can be easily modified and mapped onto different architectures in order to evaluate implementation alternatives. The target architecture as well as the mapping influences the adherence of the system to application requirements. Beside functional requirements this also includes non-functional requirements like response time of tasks or deadlines to be met.

Another important issue is the specification of tasks which can be executed in parallel. While already necessary in the context of multi-tasking, when several tasks are sharing a single processor, this becomes crucial if the application is to be mapped onto many-core architectures with hundreds of processing elements. Therefore, to be able to exploit this multitude of cores, as much application parallelism as possible must be preserved during the specification and implementation. Since OSSS is meant to serve as an executable specification, not only the structure, but also the behavior and most importantly the interaction of the parallel tasks are specified on the OSSS application layer.

In OSSS, the *Application Layer* model focuses on the logical synchronization requirements of the application. Therefore, low-level details of the communication between the application tasks are not considered here, such as the actual implementation of the communication channel, even across hardware/software boundaries. This is enabled by a concept introduced by OSSS without a direct equivalent in C++, the so called *Shared Object*. A Shared Object equips user-defined classes with specific synchronization facilities. Due to their well-defined hardware synthesis semantics, *Shared Objects* can act as a replacement for some non-synthesisable features of SystemC such as hierarchical channels, mutexes and semaphores.

Synchronization is performed by arbitrating concurrent accesses and a special feature

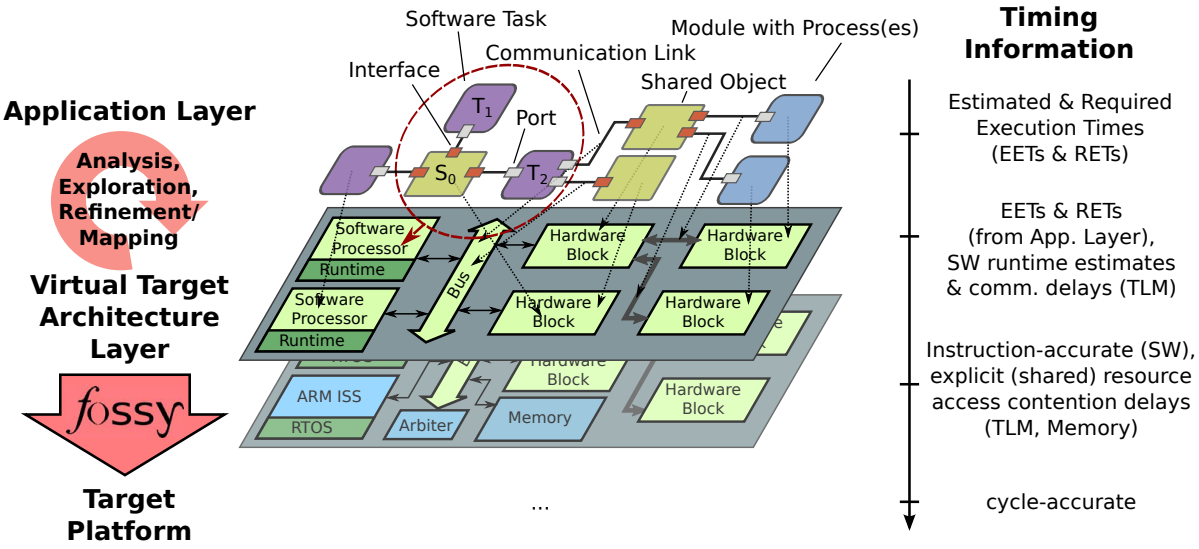


Figure 2.1: Overview of the OSSS Methodology

called *Guarded Methods*, that can be used to block the execution of a method until a user-defined condition evaluates to `true`. As a result, they are especially useful for modeling inter-task communication in parallel applications, even between hardware and software tasks. From the application's perspective, communication between tasks and *Shared Objects* is performed by direct method calls through abstract communication links. An in-depth description of the *Shared Object* concept, including several design examples, is part of the OSSS manual [13].

2.1.2 Virtual Target Architecture Layer

In a refinement step the *Application Layer* model is transformed to a so-called *Virtual Target Architecture*. This involves mapping software tasks to processor(s), hardware modules to dedicated hardware blocks, and Shared Objects to either memories or dedicated hardware, as shown in Figure 2.1. Moreover, the abstract communication links of the *Application Layer* model are mapped to specific communication infrastructure, like buses or point-to-point channels.

The host-based simulation of the *Virtual Target Architecture* provides fast, yet accurate simulation results helping the designer to perform partitioning decisions. To further improve these analysis results, especially on multi-core platforms and with shared resources, the *Virtual Target Architecture* can be refined using an ISS-based simulation approach to provide even more detailed simulation results for shared resource access delays – especially due to the consideration of delays induced by implicit memory access across shared bus topologies.

2.1.3 Remote Method Invocation

The easy and flexible mapping of the method-based communication on the *Application Layer* to a platform-specific interconnect/protocol is based on the *Remote Method Invocation* (RMI) concept. This mechanism abstracts from the platform communication infrastructure and enables the seamless refinement of specific applications on top of a predefined, yet possibly customized platform. Software Tasks should not need to be modified, when different mappings of shared resources to alternative implementations (dedicated hardware, shared memory, core/OS-local primitives) are chosen. Invoking the RMI protocol to communicate across the HW/SW boundary involves four logical and sequentially processed states [12]:

1. Serialization of the arguments for the requested method.
2. Notification of the Shared Object, sending the unique Client ID, and the requested method ID to the hardware Shared Object. The Shared Object's local scheduler arbitrates concurrent requests and grants the access eventually.
3. Sending of the arguments to the Shared Object, deserialization and execution of the requested functionality within the Shared Object.
4. An optional return value is sent back to the client (using the same, but reverse serialization mechanism)

Since multiple clients of a single Shared Object may be sharing the same processor core, the implementation of the RMI protocol requires additional runtime support by the OS or a specific driver as shown in Figure 2.2. The reason for this extended synchronization need is caused by the multi-level locking of the Shared Object in terms of a) mutually exclusive access of concurrent requests and b) the potential blocking due to the guard conditions.

If the request of a local client task is blocked by a non-granted guard condition, it still needs to release the (local) lock of the Shared Object, to avoid a (local) deadlock: A second task running on the same core as the currently blocked task would be locked out of this Shared Object, potentially never modifying the state of the Shared Object to change the guard condition currently blocking the first task.

In addition to the support of such complex synchronization requirements, a unified interface to different flavors of Shared Object implementations increases the decoupling of computation and communication and enables seamless refinement. This includes the so-called *Software Shared Objects*, where the method execution is performed within the context of the calling process directly. They can either be built on top of OS synchronization primitives (in case of all clients being managed by the same OS instance), or in terms of explicitly shared memory used from multiple cores. In the latter case, the locking mechanism needs to be implemented explicitly.

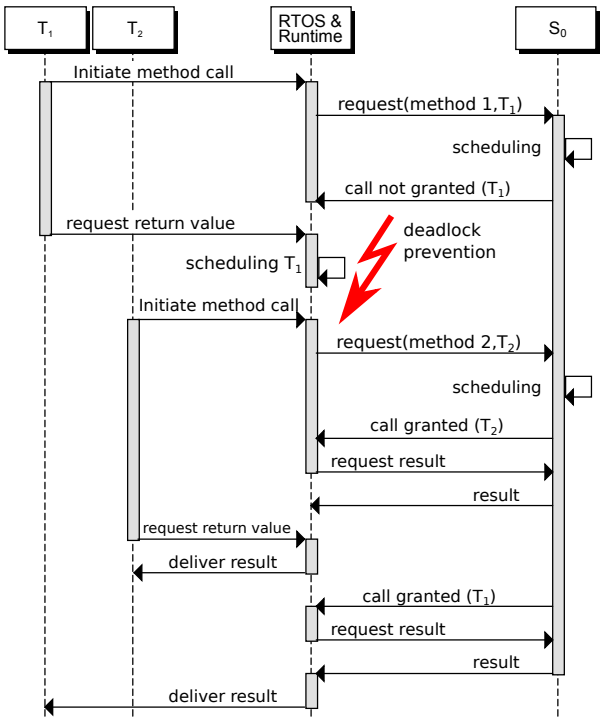


Figure 2.2: Example sequence of two Software Tasks accessing a Shared Object, following the configuration of tasks and objects in Figure 2.1.

2.1.4 Software Tasks and Execution Times

In OSSS application tasks meant to be implemented in software are modeled as *Software Tasks*. These tasks must be derived from the common base class `osss_software_task` and define their entry point in terms of a `main()` function. This function serves as executable specification, already providing the algorithmic behavior of the task. In the later refinement steps, the task's function can be cross-compiled for the target processor.

A proper modeling of software requires the consideration of its timing behavior. In OSSS, the **E**stimated **E**xecution **T**ime (EET) of a code block can be annotated within *Software Tasks* using the `OSSS_EET()` block annotation (see also [13]). The block annotation defines a duration (via an `sc_time` argument), that estimates the execution time of the following code block. This enables a flexible and accurate annotation. Granularity of the annotated blocks and thus timing behavior can be further refined. Control structures can efficiently be annotated and during simulation, the resulting execution time respects the (potentially data-dependent) control flow. Listing 2.1 exemplifies the syntax of these annotations. `OSSS_EET()` blocks must not be nested or contain any communication calls. These times are determined by either profiling or analyzing the cross-compiled code for the target processor and back-annotated to the source code [34, 29]. In addition to EETs, OSSS enables the designer to specify local deadlines for a specific code block using the **R**equired **E**xecution **T**ime annotation `OSSS_RET()`. If required, RETs can be nested at

```
while( some_condition )
// the following block has to be finished within 1ms
OSSS_RET( sc_time( 1, SC_MS ) )
{
  OSSS_EET( sc_time( 20, SC_US ) ) {
    // computation, that consumes 20µs
  }
  // estimate a data-dependent loop
  for( int i=0; i<max_i; ++i )
    OSSS_EET( sc_time( 100, SC_US ) ) {
    // loop body
  }
  if( my_condition ) {
    // communication only outside of EET blocks
    result = my_shared->get(); // see Listing 2.2
  }
} // end of RET block and loop
```

Listing 2.1: Example of estimated and required execution time annotations.

arbitrary depth. The consistency of nested RETs is checked during the simulation as well as a violation of the RETs. Unmet RETs may arise from (additional) delays caused by blocking guard conditions, or simply unexpectedly long estimated execution times (e. g. `max_i` ≥ 9 in Listing 2.1).

Formal Notation. While in the executable specification real algorithmic behavior is given, the integration with the formal analysis model abstracts from the actual internal functionality by only considering local computation nodes and explicit communication. Execution times are derived from the EET annotations. In this work, we focus on tasks with fixed activation periods and abstract activation traces.

Tasks communicate with other tasks via Shared Objects, statically bound to local *ports*. The internal behavior of a task is abstracted to a set of *activation traces*, obtained from a CDFG [1] with explicit *call nodes* for Shared Object service calls. For each possible linear[1] control flow path through the CDFG, all consecutive *data-flow nodes* are merged into the next *call node*, summing up their annotated execution times. Data-flow nodes following the last call node are merged into a single *exit node* at the end of the trace, annotated with the combined remaining execution times.

To reduce the number of traces, equivalent traces containing the same sequence of call nodes are merged into a single, *normalized trace* with combined *delay intervals* obtained from the minimum and maximum of the merged call node annotations. This leads to the following definition:

[1]For simplicity, we assume unrolled loops here. For data-dependent iteration counts, an upper bound needs to be known.

Definition 1 (Task) *Let $\mathcal{S}_i$ be a set of service symbols. An* OSSS Task *is a tuple* $\mathsf{T} = (\mathsf{P}, \pi, \mathsf{d}, \Gamma)$*, with*

1. *a set of* ports $\mathsf{P} = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$*, each representing a set of associated services* $\mathsf{p}_i \subseteq \mathcal{S}_i$*,*
2. *a period* π*, and an optional deadline* d*, and*
3. *a set* $\Gamma = \{\gamma_1, \ldots, \gamma_m\}$ *of* normalized activation traces*.*

Each activation trace $\gamma_i = \langle \gamma_{i,1}, \ldots, \gamma_{i,l_i}, \Delta_{end_i} \rangle$ *is a finite sequence of* l_i trace points $\gamma_{i,j} = \left(\mathsf{p}_{k_j}.\mathsf{s}_{q_j}, \Delta_{q_j}\right)$ *with ports* $p_{k_j} = \{\mathsf{s}_1, ..., \mathsf{s}_r\} \in \mathsf{P}$ *where* $k_j \in \{1, ..., n\}$*,* $q_j \in \{1, ..., r\}$ *and the combined WCET intervals* $\Delta_{q_j}, \Delta_{end_i} \in \mathbb{R}_0^+ \times \mathbb{R}_0^+$*.*[2]

2.1.5 Modeling and Abstraction of HW/SW Communication

For coordinating the work and to perform data exchange between tasks potentially running on different processing cores, support for inter-task communication is required, eventually even crossing the hardware/software or RTOS boundaries.

As already outlined in the previous sections, the OSSS Application Layer Model explicitly models communication and synchronization in terms of user-defined Shared Objects, which are inspired by the Protected Objects known from Ada [7]. As outlined in Section 2.1, Shared Objects provide mutual exclusive access and *Guarded Methods* to ensure deterministic behavior even with several concurrent tasks (see Listing 2.2).

As the Application Layer gives the logical perspective of the system, the execution platform itself is not explicitly modeled. All tasks and Shared Objects are implicitly mapped on exclusive processing elements. Consequently, only logical contention on Shared Objects can be observed here. This contention can be caused either by concurrent accesses to the same Shared Object or due to blocking times induced by method guards.

```
// guard definition ( <name>, <condition> )
OSSS_GUARD( not_empty, cnt_items > 0 );

// guarded method declaration of int get() – blocking, until not_empty guard holds
OSSS_GUARDED_METHOD( int, get, OSSS_PARAMS(0), not_empty );
```

Listing 2.2: Example of a Shared Object's guarded method.

Starting from the OSSS Virtual Target Architecture, additional delays can be induced by the additional resource sharing of tasks and Shared Objects mapped to the same processing elements. These delays are exposed by the more detailed simulations of the models on the lower layers.

On the final platform, the implementation of a Shared Object depends on its mapping. It can either be mapped onto dedicated hardware or a software runtime system (as

[2] Δ_{end_i} is the combined WCET interval at the end of T in trace γ_i.

described in Section 2.1.6). For the implementation of hardware/software communication at least the communication interfaces of a Shared Object need to be accessible for both communication parties.

Formal Notation. We now introduce a notation which abstracts from the details implicitly contained in the C++-based model description. Following the object-oriented programming paradigm, a Shared Object provides a set of services grouped by interfaces to their clients.

Definition 2 (Shared Object) *An OSSS* Shared Object *is a tuple* $\mathsf{SO} = (\mathcal{S}, \mathcal{I}, \Delta)$*, where*

1. $\mathcal{S}$ *is a set of symbols, representing the provided* services*.*
2. $\mathcal{I} = \{\mathsf{IF}_0, \ldots, \mathsf{IF}_k\}$ *is a set of* interfaces*, where each* $\mathsf{IF}_i \subseteq \mathcal{S}$ *denotes a subset of services.*
3. $\Delta : \mathcal{S} \rightarrow \mathbb{R}_0^+ \times \mathbb{R}_0^+$ *provides a (worst case) execution time estimation interval for each service.*

The internal state of the Shared Object is not modeled explicitly for now, and only worst-case behavior is considered. A future extension will explicitly introduce (meta) states of the SO and extend the behavior abstraction to cover abstract state changes as well, enabling the consideration of guard states in the abstract model.

2.1.6 Modeling Multitasking in OSSS

Before refining the model to the final implementation platform, several virtual prototyping simulations are supported by OSSS. On the Virtual Target architecture, a host-based simulation of the software tasks is performed. In order to support the accurate simulation of multiple (software) tasks being mapped to the same processing element (CPU core), the effects of local scheduling, usually performed by operating systems or some local software run-time, need to be considered.

The approach to model multitasking in OSSS is not meant to directly represent existing real-time operating system (RTOS) primitives, since the Application Layer model has a well-known structure. The *Software Tasks* (see Section 2.1.4) in OSSS are meant to *run* on-top of a rather generic, lightweight run-time system (see Figure 2.1), where the synchronization and inter-task communication is modeled with *Shared Objects* (as introduced in Section 2.1.5), similar to the modeling with OSSS in the hardware domain. This enables a seamless specification environment, where the same concepts are used for both, hardware *and* software on the Application Layer (see Section 2.1.1). The flexible timing annotation mechanism (Section 2.1.4) enables high simulation performance, since the synchronization overhead with the SystemC kernel can be minimized, as shown in [14].

2.1.6.1 Abstraction of Run-time System

The basis of the OSSS software run-time simulation model is an OSSS RTOS abstraction class. This predefined library element handles the time-sharing of a single processor by several *Software Tasks*, which are bound to this OS instance. A specific scheduling policy can be specified for the tasks. Several frequently used scheduling policies are already provided by the OSSS library, most notably static priorities (preemptive and cooperative), time-slice based round-robin, earliest-deadline first, and rate monotonic. Additionally, arbitrary user-defined scheduling policies can be added as long as they implement an abstract interface class. The RTOS overhead of context switches and execution times of scheduling decisions can be annotated as well.

With this set of basic elements, the behavior of the real RTOS on the target platform can be modeled. Task synchronization is not part of the modeling elements, since the inter-task communication is modeled using *Shared Objects*.

On the final platform, the software implementation of a Shared Object for inter-task communication has to be integrated with the OSSS software run-time. The guard mechanism for *Shared Objects* which are mapped onto a software run-time system, as well as the required locking primitives for mutual exclusion are then directly implemented using existing locking mechanisms of the underlying RTOS (e.g. mutexes).

2.1.6.2 Software Execution Times and Multitasking

A proper modeling of software multitasking requires the consideration of time consumption of the modeled tasks. For this purpose we extend the execution time annotation described in Section 2.1.4 for the extended Software Tasks and allow code block annotations inside methods of *Shared Objects*.

Again, RETs enable the designer to specify local deadlines for a specific code block. When using software multitasking this is especially useful in combination with inter-task communication calls or preemptive scheduling policies. Unmet RETs can now also arise from the choice of the scheduling policy.

In order to ensure that only one *Software Task* is active at any time during the simulation, the different tasks have to be synchronized with the RTOS abstraction, which then assigns the tasks according to its scheduling policy.

In order to support a preemptive scheduling policy, this implicit synchronization is performed by the abstract run-time system. For every EET block, the run-time advances the SystemC time for the current task by the annotated time *and* the additional delay due to preemptions of the current task by other (e.g. higher priority) tasks during this period. The same technique has been proposed in the ROM (Result Oriented Modeling) approach in [26].

2.2 Analysis Model - Syntax and Semantics

In the following, we will briefly recapitulate the component- and contract-based approach proposed in [9], which forms the basic underlying concepts for representing models located

in the different cells of the matrix shown in Figure 1.2. Afterwards we characterize a relevant class of models in the context of real-time systems that we call task networks. We will discuss how these task networks integrate with the former component models. Given a task network embedded in some component model, we instantitate the framework of real-time interfaces [30] in order to analyze its timing behavior when allocating it to elements of the technical platform architecture. Finally, we present an abstraction technique to capture assumptions about a technical platform architecture. Based on these assumptions, deployment of a task network embedded in a component model can be analyzed in a compositional manner. This technique will be useful in the later mapping to the OSSS methodology, as the assumptions can be regarded as characterization of a set of possible platform architectures and their configuration. Thereby we constrain the design space which can then be explored using the OSSS simulation capabilities.

2.2.1 Components and Contracts

Alongside the two-dimensional design-space briefly introduced in Section 1, the meta model proposed in [9] is based on a component concept similar to those found in e.g. SysML [21], EAST-ADL [2] and AUTOSAR [3]. A component has a well defined syntactical interface in terms of its ports. Components can be assembled forming so called compositions, which are again regarded as components. A speciality of this component notion is its strict separation of specification of different behavioral aspects and a realization of these aspects. Aspects specifications are assumed to be formulated using contracts promoting an assume/guarantee style. That means a contract explicates assumptions about the environmental context of a component under which its it behaves as guaranteed by the contract. Figure 2.3 depicts these main features of the component notion. The task network class of models presented next can be regarded as models

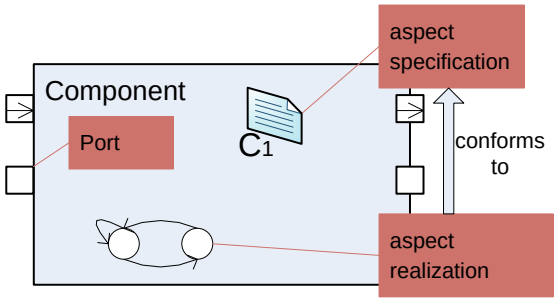


Figure 2.3: Component concept, aspect specification and realization

representing the realization of the timing aspect of a component. In this work we are particularily interested in the deployment phase. That is the mapping of components located in the logical perspective containing task networks onto elements of a technical platform. Therefore, we will also define a real-time model as a task network whose elements are allocated over a set of resources, with these resources representing the techical architecture.

2.2.2 Task Networks

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 [32]. 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 for the set Σ. Additionally, each event might be further delayed by a jitter j.

Definition 3 *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* $\tau = (p^I_\tau, \Gamma, P^O_\tau)$ *where* $p^I_\tau \in P$ *is an input port,* $\emptyset \neq P^O_\tau \subseteq P$ *a set of output ports, and*

$$\begin{aligned} \Gamma =& \Sigma(p^I_\tau) \to \Psi \\ \Psi =& \bigcup\nolimits_{\emptyset \neq Q \subseteq P^O_\tau} \{\psi : Q \to (\Sigma(Q) \times \mathbb{N}^+ \times \mathbb{N}^+) \mid \\ & \qquad \psi(p_o) = (\sigma, \delta^-, \delta^+) \implies \sigma \in \Sigma(p_o)\} \end{aligned}$$

where Γ *maps input events arriving at the input port* p^I_τ *to an output specification* $\psi \in \Psi$ *that maps output ports to output events and execution intervals.* ◇

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 simplicity of the definitions.

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 4 *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* $\tau \in \mathcal{T}$. ◇

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 notion of real-time interfaces for verification of these real-time models can cope with a broad range of scheduling schemes.

2.2.3 Real-Time Interfaces

The notion of real-time interfaces defined in [6] 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, like shown in Figure 2.5 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 2.5 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. [23]) by lower and upper arrival curves* $\eta^-(\Delta t)$ *and* $\eta^+(\Delta t)$.

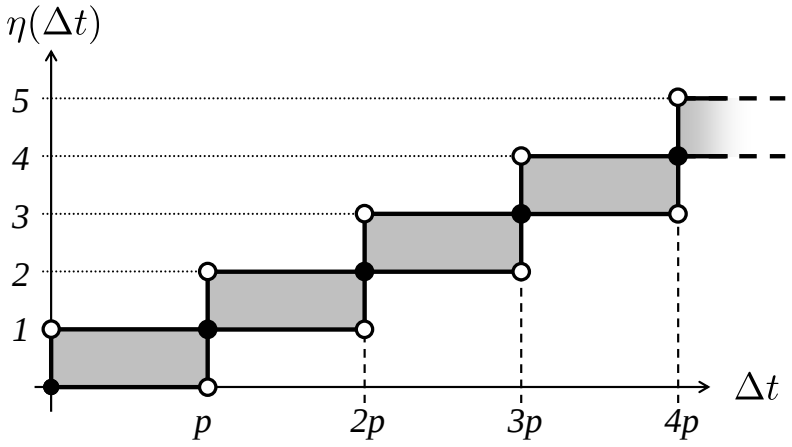


Figure 2.4: Arrival curves of periodic events

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 elaborated 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, [6] 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 over tuples of symbols as defined in [30]. 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.

While [6] 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 a set of tasks is allocated, which is represented by the alphabet Σ_r.

Definition 5 *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 $\Sigma_K = \prod_{k \in K} \Sigma_k$, $L_K \subseteq \Sigma_K^\omega$. $\diamond$

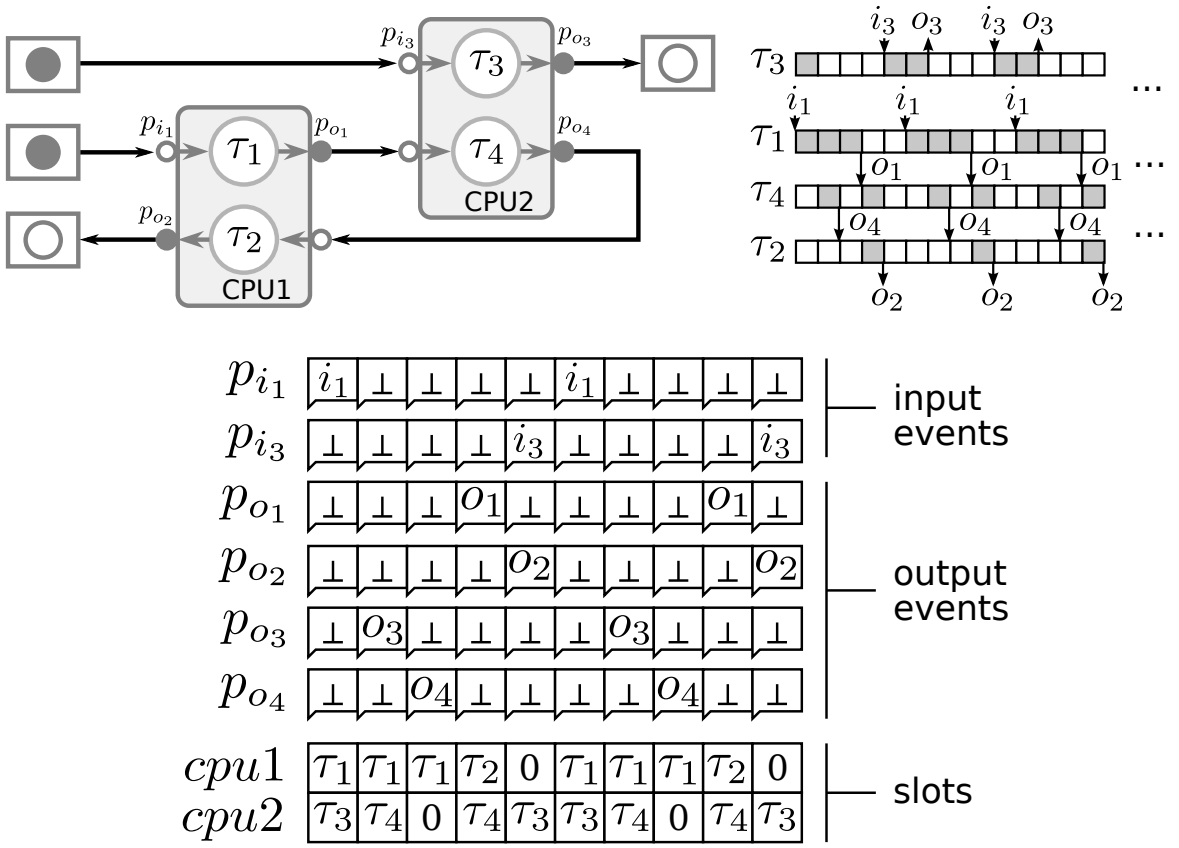


Figure 2.5: System example (top) and exemplary trace-extract (bottom) of the corresponding interface.

Example 2 *Focus on the example system depicted on the left hand side of Figure 2.5. 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 by a fixed priority policy with preemption, where tasks τ_1 and τ_3 have high priority on their respective CPU. The delay of the task-chain $\tau_1 \rightarrow \tau_4 \rightarrow \tau_2$ 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 2.5. 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 $K = P \cup R$, $P = \{p_{i_1}, \ldots, p_{i_4}, p_{o_1}, \ldots, p_{o_4}\}$ 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 2.5, which corresponds to the previously discussed scheduling scenario. Intuitively,*

each port in a system has its own event-tape in the interface, as well as 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 synchronization of the behaviour.

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 elaborated. 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 6 *Let be $N = \{1, ..., n\}$, 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 7 *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, vice versa 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 2.5. 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. Hence given $I_1 \parallel I_2 = (K, \Sigma_K, L_K)$, we partition $K = P_{in} \uplus P_{out} \uplus R$, where

- $P_{out} = P_{1_{out}} \cup P_{2_{out}}$
- $P_{in} = (P_{1_{in}} \cup P_{2_{in}}) \setminus P_{out}$
- $R = R_1 \cup R_2$

In summary, composition of interfaces is merely inverse projection followed by intersection of the languages. Though this composition operation is well defined for arbitrary interfaces, it turns out that for some cases its resulting language imposes unintended restrictions on the behavior of the constituting interfaces. Consider the following example: Assume given simple interfaces I_1 and I_2 each containing exactly one task with an execution time of 3 executed periodically with a rate within the interval $[5, 6]$. The composition $I_1 \parallel I_2$ would have a non-empty language representing its legal schedules. These schedules would demand that the tasks of both interfaces are activated at a period of exactly 6. However, the activation behavior is controlled by the evironment of $I_1 \parallel I_2$ and not by the interface or its tasks. As we do not know this environment yet, we must preserve the assumptions of both interfaces about that evironment explicated by means of the activation behavior of the tasks. So we have restricted behavior in an illegal way. As a second example consider two simple interfaces I_1 and I_2 each containing exactly one task. Assume both tasks are executed with a period of 5 and have an execution time interval $[2, 3]$. Again the composition $I_1 \parallel I_2$ restricts behavior in an illegal way, as the language of $I_1 \parallel I_2$ representing legal schedules does not contain traces where both tasks have an execution time of 3. The fact that both tasks had an execution time within a given interval is usually a result of either an uncertainty about the exact execution time or the execution time could be dependent on the input values of the task. No matter what the reason was, the execution time interval must be treated as a non-deterministic choice about how long the task needs to carry out its computations. Thus, similarily to activation behavior, execution times of tasks cannot be "controlled" by an interface, which means we must not restrict them. Thus, we need to define such illegal restrictions and based on that a notion of *composability* of real-time interfaces. This notion is defined such that given two composable interfaces I_1 and I_2, their composition $I_1 \parallel I_2$ is well defined in the sense that behavior is not restricted in an unintended illegal way. In particular, composability ensures that all possible activation behaviors of composable interfaces are preserved under composition, as well as their maximal execution demands. For the second condition we need means to "compare" the execution demands of words

of an interface language. This allows us to characterize what kind of words must be preserved under composition. Hence we define an order on slot words:

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

This defines partial orders $(\Sigma^\omega, \leq)$ and $(\Sigma_K^\omega, \leq)$ on slot words. Obviously, 0^ω is the bottom element, $(0, \ldots, 0)^\omega$ respectively. A word ω' precedes ω if 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' (i.e. $\sigma'_i \neq 0$), is used in the same way in ω. Based on this order on slot words, we define the following order on slot languages over Σ_K^ω:

Definition 9 *Given two languages* $L_K, L'_K \subseteq \Sigma_K^\omega$, *we define* $L'_K \sqsubseteq L_K$, *if and only if* $\forall \omega'_K \in L'_K : \exists \omega_K \in L_K : \omega'_K \leq \omega_K$. ◇

This defines a pre-order $(\mathcal{P}(\Sigma_K^\omega), \sqsubseteq)$ on slot languages, as $L_K \sqsubseteq L'_K$ and $L'_K \sqsubseteq L_K$ does not necessarily imply equivalence of L_K and L'_K. 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 due to the existential quantification we can add arbitrary words to L_K without breaking the order $L'_K \sqsubseteq L_K$. On the opposite, given $L'_K \sqsubseteq L_K$, there may exist a subset $X \subseteq L_K$, such that $L'_K \sqsubseteq X \sqsubseteq L_K$. We are interested in a particular unique subset of a slot language $L_K \subseteq \Sigma_K^\omega$, containing only those words from L_K with maximal execution demands. We denote this subset by $\widehat{L_K}$, a *maximal language subset*, and define it as follows:

Definition 10 *Given a slot language* $L_K \subseteq \Sigma_K^\omega$, *the subset* $\widehat{L_K} \subseteq L_K$ *is given by*

$$\widehat{L_K} = \{\omega_K \in L_K \mid \forall \omega'_K \in L_K : \omega_K \preceq \omega'_K \implies \omega_K = \omega'_K\}$$

◇

Indeed the subset $\widehat{L_K}$ characterized by Definition 10 is unique and maximal with respect to the order $\sqsubseteq$ as stated by the following proposition:

Proposition 1 *For every* $L_K \subseteq \Sigma_K^\omega$ *the subset* $\widehat{L_K} \subseteq L_K$ *is unique and maximal, i.e.* $\forall L'_K \subseteq L_K : L'_K \sqsubseteq \widehat{L_K}$. ◇

Applying the relation from Definition 9 to the set of maximal languages, we obtain the partial order $(\widehat{\mathcal{P}(\Sigma_K^\omega)}, \sqsubseteq)$. Intuitively, the maximal subset $\widehat{L_K}$ from Definition 10 deletes all words from L_K, whose slot usage is "dominated" by another word in L_K. The following proposition directly follows from Definition 9 and Definition 10:

Proposition 2 *Given slot languages* $L'_K, L_K \subseteq \Sigma_K^\omega$ *and* $L'_K \sqsubseteq L_K$, *it holds that* $L'_K \sqsubseteq \widehat{L_K} \sqsubseteq L_K$. ◇

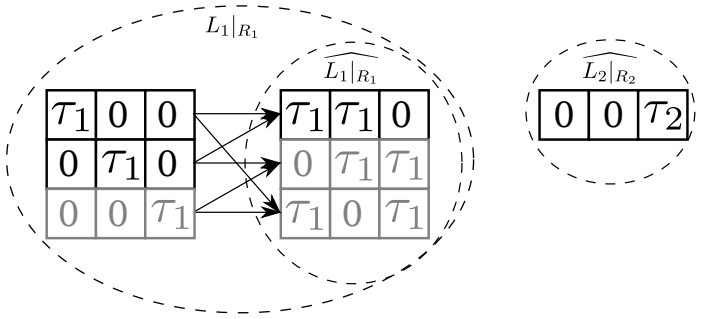


Figure 2.6: Illustration of Definition 11

We are now ready to define composability of real-time interfaces such that under composition $I_1 \parallel I_2$, behavior of the constituting interfaces is not restricted in an illegal way:

Definition 11 *Two real-time interfaces I_1 and I_2 are composable if and only if their input port sets $P_{1_{in}}$ and $P_{2_{in}}$ are disjoint, as well as their output port sets $P_{1_{out}}$ and $P_{2_{out}}$ and the following condition holds:*

$$\forall a \in L_{12}|_{P_{1_{in}} \cup P_{2_{in}}} : proj^{-1}(\Sigma_R, \Sigma_{R_1})(\widehat{L_1(a)|_{R_1}}) \cap proj^{-1}(\Sigma_R, \Sigma_{R_2})(\widehat{L_2(a)|_{R_2}}) \neq \emptyset$$

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

The condition from the above definition addresses both of the discussed kinds of illegal behavior restriction. Basically, it requires that for *all activation behaviors* of both interfaces their potentially resulting *maximal* execution demands are schedulable, i.e. intersection of the maximal slot languages of the interfaces is not empty. Ensuring that the maximal execution demands of the interfaces are schedulable for all activation patterns, implies that schedulability is also given for the cases where an interface demands less slots. Consider the following example illustrated in Figure 2.6: An interface I_1 consists of a single task with an execution time interval of $[1, 2]$ and a period of 3. The slot language of I_1 is shown in the figure together with its subset $\widehat{L_1|_{R_1}}$. The directed arrows point from a word, which is "less" according to Definition 8 to a "greater" word. If we now consider another interface I_2, whose language $\widehat{L_2|_{R_2}}$ is shown on the right hand side of Figure 2.6, we observe that the words $\omega_1 = [\tau_1\tau_1 0]^\omega$ and $\omega_2 = [00\tau_2]^\omega$ can be scheduled, and $[\tau_1\tau_1\tau_2]^\omega$ is part of the slot language $L_{I_1 \parallel I_2}|_R$ of the composition. And since the words $\omega_1' = [\tau_1 00]^\omega \leq \omega_1$ and $\omega_1'' = [0\tau_1 0]^\omega \leq \omega_1$ can both be scheduled with the word ω_2, we also have $[\tau_1 0\tau_2]^\omega \in L_{I_1 \parallel I_2}|_R$ and $[0\tau_1\tau_2]^\omega \in L_{I_1 \parallel I_2}|_R$. Note: The fact that we cannot schedule all words from $\widehat{L_1|_{R_1}}$ with ω_2 is not a problem, because those words represent alternative schedules of I_1 with maximal demands. It is therefore sufficient to find at least one word in the intersection of $\widehat{L_1|_{R_1}}$ and $\widehat{L_2|_{R_2}}$.

The following proposition addresses the potential restriction of activation behavior. For two composable interfaces I_1 and I_2, their composition does not restrict activation

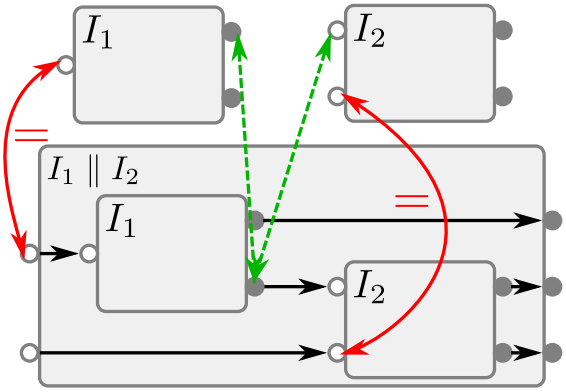


Figure 2.7: Illustration of Proposition 3

behavior observable at input ports $P_{i_{in}}$ of each interface I_i, that are still open in the composition (i.e. not connected to an output port of another interface).

We are only concerned with open input ports, because if output ports of an interface (like I_1 in Figure 2.7) are connected to input ports of another interface (like I_2), then interface I_1 plays the role of an environment for I_2. Hence, as we know the evironment of I_2, we can unify connected ports in the composed interface. The behavior observable at those ports in the composition $I_1 \parallel I_2$ can indeed be a subset of the behavior that was assumed at the corresponding input ports in the interface I_2.

Proposition 3 *Given two composable real-time interfaces I_1 and I_2 and their composition $I_1 \parallel I_2$, it holds that:*

$$L_i|_{P_{i_{in}} \cap P_{12_{in}}} = proj(\Sigma_K, \Sigma_{K_i})(L_{12})|_{P_{i_{in}} \cap P_{12_{in}}}$$

$\diamond$

Proof: We proof Proposition 3 by contradiction. Given two composable interfaces I_1 and I_2, suppose that $L_i|_{P_{i_{in}} \cap P_{12_{in}}} \neq proj(\Sigma_K, \Sigma_{K_i})(L_{12})|_{P_{i_{in}} \cap P_{12_{in}}}$ holds for at least one of I_1 and I_2. To ease exposition, we continue the proof for I_1, because the same arguments holds for I_2. As composition of interfaces is language *intersection* (after inverse projection), we have $L_1|_{P_{1_{in}} \cap P_{12_{in}}} \supset proj(\Sigma_K, \Sigma_{K_1})(L_{12})|_{P_{1_{in}} \cap P_{12_{in}}}$. Remember that we consider interfaces over tuples of symbols and require the input port sets $P_{1_{in}}$ and $P_{2_{in}}$ to be disjoint, as well as their output port sets $P_{1_{out}}$ and $P_{2_{out}}$. So the reason for "loosing" words in the composition due to intersection must be that the normalization function maps resources of I_1 and I_2 to the same resource in $I_1 \parallel I_2$, i.e. $R_1 \cap R_2 \neq \emptyset$. Hence, it follows that $\exists a \in L_{12}|_{P_{1_{in}} \cup P_{2_{in}}} : proj^{-1}(\Sigma_R, \Sigma_{R_1})(L_1(a)|_{R_1}) \cap proj^{-1}(\Sigma_R, \Sigma_{R_2})(L_2(a)|_{R_2}) = \emptyset$. This contradicts Definition 11, and therefore I_1 and I_2 would not be composable. $\square$

Definition 12 *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.* ◇

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.* ◇

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 $I' \parallel J \preceq I \parallel J$.* ◇

2.2.4 Adding Contracts

As in [30] 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 [5]. 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 a components ports P. 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 consider compositions of components, it becomes important to care about the ports contracts talk about. Hence we define a contract over a set of ports as a tuple $C = (P, \Sigma_P, A, G)$ where $A, G \subseteq \Sigma_P^\omega$.

Definition 13 *[5] 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).* ◇

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 [5]. 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 14 *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. ◇

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 compositional also for the notion of real-time 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 $C_1 \parallel C_2$.* ◇

2.2.5 Resource Segregation

While real-time interfaces are powerful enough to cope with complex designs and component integration scenarios like depicted in Figure 2.5, the refinement relation involves complex language inclusion checks. Moreover, the details of all components and their tasks must be known in order to compose them. Therefore, we introduce an abstraction for a real-time interface consisting of multiple tasks that we call *segregation property*. Segregation properties were first proposed in [22] and were based on the interface definition presented in [6]. In this work we present a revised notion of segregation properties based on the extended real-time interface definitions found in Section 2.2.3 and [30]. These segregation properties will be defined such that composability of segregation properties ensures composability of their interfaces (according to Definition 11), respectively. That means given two interfaces I_1 and I_2 and segregation properties B_{I_1} and B_{I_2}, we look for a composition operator $\parallel$ and a simple property φ, such that

$$B_{I_1} \parallel B_{I_2} \models \varphi \implies I_1 \parallel I_2 \text{ is schedulable}$$

Recall, that an interface I consists of a language over tuples of symbols describing a set of legal schedules. It represents for the activation patterns (behavior observable at the input ports) of its tasks a set of possible discrete slot allocations under which the tasks can be executed successfully. A segregation property B_I for an interface abstracts from the tasks of the interface, and only exposes a set of possible slot reservations for

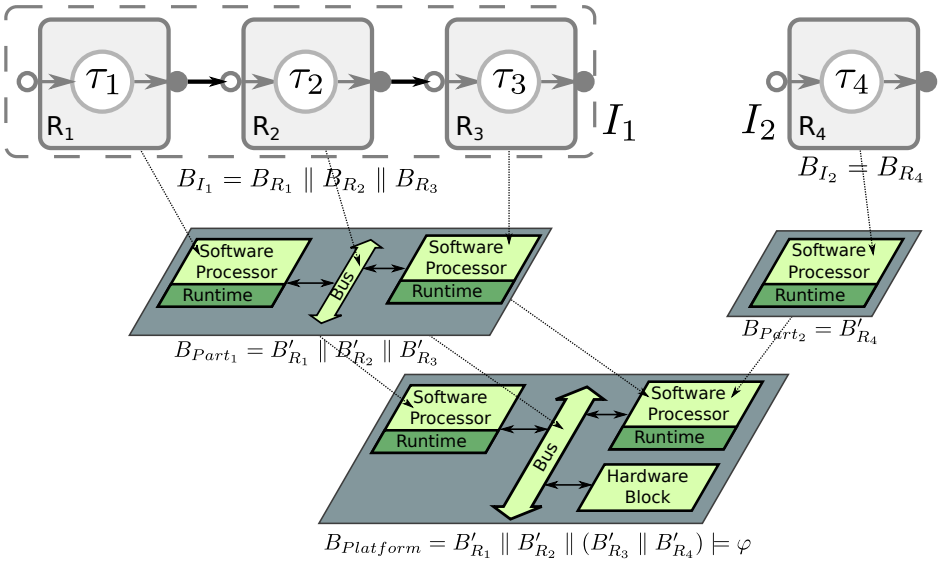


Figure 2.8: Compositional deployment scenario using Segregation Properties

which the interface is schedulable for *all* its activation patterns. Note that a segregation property for an interface indeed may contain more available slots than are used by the respective interface.

The basic idea is that composition of the 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}. The property φ states that at least one such non-conflicting slot reservation exists, i.e. the set of slot reservations defined by $B_{I_1} \parallel B_{I_2}$ is not empty.

We now define *segregation property* of an interface and a composition operation. We use that composition operation to derive a sufficient condition for interface composability in the sense of Definition 11 based on their segregation properties. Further, we provide a refinement notion that allows us to relate segregation properties. Thereby, scenarios like depicted in Figure 2.8 can be supported, where deployment of different applications captured by interfaces I_1 and I_2 can be analyzed compositionally by means of their segregation properties. The resources of the target platform shown at the bottom Figure 2.8 of would be partitioned where each of these different partitions are characterized by segregation properties B_{Part_1} and B_{Part_2}. Timing behavior of the applications can then be analyzed based on their respective segregation properties, and refinement of the platform segregation properties B_{Part_1} and B_{Part_2} by the application segregation properties B_{I_1} and B_{I_2} ensures that both applications can be deployed on the partitioned resources of the platform without invalidating their timing guarantees.

Definition 15 *Let I_K be an interface, and let $B \subseteq \Sigma_R^\omega$ be a slot reservation language over $R \subseteq K$. B is a* segregation property *for I_K iff*

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

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

That means B is a segregation property for I_K, if for *all* its possible activation patterns *each* word in B is "dominating" at least one of the maximal execution demands resulting from the activation pattern. In other words: B is a segregation property for I_K, if I_K is schedulable for *all* its possible activation patterns under *all* slot reservations defined by B.

For the composition of slot reservations languages we do not need a special operation. Instead the composition operation defined for interfaces (see Definition 7) directly applies to slot reservations languages. The condition for composability of slot reservation languages is rather simple:

Definition 16 *Two slot reservation languages $B_1, B_2 \subseteq \Sigma_R^\omega$ are composable if and only if:*

$$B_1 \parallel B_2 \neq \emptyset$$

$\diamond$

This notion of composability of slot reservation languages is the desired simple property φ mentioned at the beginning of this section. If two composable slot reservations languages B_1 and B_2 are segregation properties for some interfaces I_1 and I_2, then $B_1 \parallel B_2$ is a segregation property for both interfaces modulo projection as stated by the following proposition:

Proposition 4 *Given segregation properties B_{I_1} for interface I_1 and B_{I_2} for interface I_2, then $B_{I_1} \parallel B_{I_2} \neq \emptyset \implies$*

- *$proj(\Sigma, \Sigma_{R_1})(B_{I_1} \parallel B_{I_2})$ is a segregation property for I_1*
- *$proj(\Sigma, \Sigma_{R_2})(B_{I_1} \parallel B_{I_2})$ is a segregation property for I_2*

$\diamond$

Proof: The proof follows directly from Definition 15 and Definition 7. According to Definition 15, B is a segregation property for interface I, if the condition $\forall a \in L_K|_{P_{in}} : \exists \omega \in \widehat{L_K(a)|_R} : \omega \leq b$ applies to *all* $b \in B$. Hence, if B is a segregation property for I, all non-empty subsets of B are also segregation properties for I. Following Definition 7, composition is merely inverse projection followed by language intersection. Thus, projecting back onto the resource alphabet Σ_{R_j} of segregation property B_{I_j} results in a non-empty language, which is a subset of B_{I_j}, and must therefore also be a segregation property for I_j. $\square$

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

Lemma 5 *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 it holds: $B_1 \parallel B_2 \neq \emptyset \implies$*

$$\forall a \in L_{12}|_{P_{1_{in}} \cup P_{2_{in}}} : proj^{-1}(\Sigma_R, \Sigma_{R_1})(\widehat{L_1(a)|_{R_1}}) \cap proj^{-1}(\Sigma_R, \Sigma_{R_2})(\widehat{L_2(a)|_{R_2}}) \neq \emptyset$$

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

That means, composability of segregation properties (see Definition 16) of interfaces implies the condition for composability of their interfaces (see Definition 11).

Proof: For the proof we make use of the same shortcut used in some of the definitions above and write $L(a)$ where $L(a) = \{\omega \in L \mid \omega|_{P_{in}} = a\}$. Choose segregation properties B_1 and B_2, such that $B_1 \parallel B_2 \neq \emptyset$. It follows from Definition 7 that $\exists b_1 \in B_1 : \{b_1\} \parallel B_2 \neq \emptyset$. From Definition 15 we know that $\forall a \in L_1|_{P_{in}} : \exists \omega \in \widehat{L_1(a)|_{R_1}} : \omega \leq b_1$. So for all activation patterns of I_1 a schedule ω with maximal execution demands exists, that uses less slots than provided by b_1. Hence it follows that $\forall a \in L_1|_{P_{in}} : \widehat{L_1(a)|_{R_1}} \parallel B_2 \neq \emptyset$. The same arguments can be applied when choosing some word $b_2 \in B_2$, allowing us to substitute B_2. As input ports of I_1 and I_2 are pair-wise disjoint, we finally get $\forall a \in L_{12}|_{P_{1_{in}} \cup P_{2_{in}}} : proj^{-1}(\Sigma_R, \Sigma_{R_1})(\widehat{L_1(a)|_{R_1}}) \cap proj^{-1}(\Sigma_R, \Sigma_{R_2})(\widehat{L_2(a)|_{R_2}}) \neq \emptyset$, which is the condition for composability of interfaces I_1 and I_2. □

Having defined segregation properties and their composition, we now discuss a refinement notion. Remember that Definition 15 defines a segregation property for an interface I as a slot reservation language B, such that I is schedulable for *all* its activation patterns under *all* slot reservations defined by B. From this definition we conclude that given a segregation property B, *any non-empty subset* $B' \subseteq B$ is also a segregation property for interface I. So in particular every $b \in B$ is a segregation property for I. Further, if the interface I is schedulable under a slot reservation b, then it will also be schedulable under any other slot reservation $b' : b \leq b'$ (see Definition 8). In other words: We can always reserve more slots for interface I without impact on its schedulability. These observations lead us to the following definition for refinement of slot reservation languages:

Definition 17 *Given two slot reservation languages $B, B' \subseteq \Sigma_R^\omega$, we say B' refines B or 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 from Definition 9, the refinement relation for slot reservations is a pre-order $(\mathcal{P}(\Sigma_R^\omega), \preceq)$, as mutual refinement $B' \preceq B$ and $B \preceq B'$ not necessarily implies equivalence of B and B'. Intuitively, $B' \preceq B$ means that *all* slot reservations $b \in B$ "dominate" at least one slot reservation $b' \in B'$. In other words: *All* slot reservations $b \in B$ must make at least the same guarantees about reserved slots (possibly reserving even more slots) than some slot reservation $b' \in B'$. Thus, if B is a segregation property for interface I, its slot reservation language can be abstracted and every $B \preceq B''$ is also a segregation property for I. This definition also captures the trivial abstraction of B by any non-empty subset $B'' \subseteq B$. Further, this notion allows independent refinement of slot reservation languages as stated by the following lemma:

Lemma 6 *Given two composable slot reservation languages $B_1, B_2 \subseteq \Sigma_R^\omega$, i.e. $B_1 \parallel B_2 \neq \emptyset$. Then $B_1' \preceq B_1$ and $B_2' \preceq B_2$ implies $B_1' \parallel B_2' \neq \emptyset$.* ◇

Proof: According to Definition 7 all $b \in B_1 \parallel B_2$ are words resulting from pairs $b_1 \in B_1$, $b_2 \in B_2$, where each slot in b is either used by b_1 or b_2 but not both. Following Definition 17 $b_1' \in B_1'$, $b_2' \in B_2'$ exist, where $b_1' \leq b_1$ and $b_2' \leq b_2$. Thus, b_1' and b_2' can be composed and $B_1' \parallel B_2'$ must contain at least one word. □

2.2.5.1 Resource Segregation Patterns

There have been considerable studies on compositional real-time scheduling frameworks [31, 33, 28, 16, 10]. These studies define interface theories for components abstracting the resource requirement of a component by means of demand functions [31, 33], bounded-delay resource models [16], or periodic resource models [28, 10]. 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. Contrary to those approaches, our real-time interfaces and resource segregation are based on ω-regular languages. That means, the approach can for example be employed in automata-based model-checking frameworks. In addition this approach is not bound to specific task and resource models, like periodic or bounded delay. In this section we review the classes of periodic resource models, proposed by I. Lee et. al., and discuss how our approach of real-time interfaces and segregation properties relates with the frameworks presented in [28, 10]. We will see that our approach is able to capture the models considered in these frameworks, and thus results established in these frameworks also apply in our setting.

Both frameworks are based on the concepts of demand bound functions $dbf(\Delta)$ and supply bound functions $sbf(\Delta)$. The function $dbf(\Delta)$ characterizes the maximal processing demand of a real-time component within any interval of length Δ. The function $sbf(\Delta)$ characterizes the minimal processing power provided by the resource in any time interval of length Δ. The real-time component is considered to be schedulable, if $\forall\Delta : dbf(\Delta) \leq sbf(\Delta)$. Note, that the concept of service curves known from real-time calculus [8] is comparable with these frameworks, as described in [33]. In [28] a *Periodic Resource Model* is presented and in [10] an *Explicit Deadline Periodic Resource Model* (EDP) is presented. Both models are used to create compositional hierarchical scheduling frameworks. In both frameworks a component is a set of tasks scheduled under a specific strategy. The total resource demand of a component to schedule all its tasks is expressed as a demand bound function $dbf(\Delta)$. The resource models are used to capture the amount of resource allocations of a partitioned resource, which is formally expressed as a supply bound function $sbf(\Delta)$. If a component is schedulable under the considered partitioned resource (defined by the resource model), i.e. $dbf(\Delta) \leq sbf(\Delta)$, then the resource model can be transformed into a task and components can be composed hierarchically. Thus, the composition problem is reduced to the abstraction problem.

The periodic resource model $\Gamma = (\Pi, \Theta)$ characterizes a partitioned resource that repetitively provides Θ units of resource with a repetition period Π. The EDP resource model $\Omega = (\Pi, \Theta, \Delta)$ is an extension of the periodic resource model. It characterizes a partitioned resource that repetitively supplies Θ units of resource within Δ time units, with Π the period of repetition. Keeping in mind the idea of transforming a resource model into a task at the next level of the hierarchy, the relation between both models becomes clear: A periodic resource model $\Gamma = (\Pi, \Theta)$ is the EDP model $\Omega = (\Pi, \Theta, \Pi)$ (cf. [10]). Therefore, in the following we focus on EDP resource models.

Real-time Component Model: A real-time component is defined as $C = \langle\{C_1, \ldots, C_n\}, S\rangle$, where C_i is either another real-time component or a sporadic task. A sporadic task is defined by a tuple $\tau = (p, e, d)$, where p is a minimum separation time, e the execution time of the task and d a deadline relative to the release of task τ. It holds that $e \leq d \leq p$. The workload $C_1, \ldots, C_n$ is scheduled under strategy S that is either RM (rate monotonic), DM (deadline monotonic) or EDF (earliest deadline first). The *resource demand* of a component is then the collective resource demand of its tasks under its scheduler S. The *demand bound function* [18, 4] $dbf_C(\Delta)$ characterizes the maximum resource demand for a task set in any given time interval of length Δ.

In our framework real-time components translate into interfaces, where each interface I is either a composition of interfaces $I = I_1 \parallel \ldots \parallel I_n$ or an "atomic interface" in case of a single sporadic task. Given a task $t = (p, e, d)$, the slot language of the corresponding interface is $L_{I_t}|_R = 0^{<p-1}[(t^e \,|||\, 0^{d-e})0^{p-d}]^\omega$, where $u \,|||\, v$ denotes all possible interleavings of the finite words u and v. Given a component $C = \langle\{C_1, \ldots, C_n\}, S\rangle$, then the condition $I_{C_1} \parallel \ldots \parallel I_{C_n} \neq \emptyset$ determines whether the component is schedulable at all under some scheduling strategy S. Now consider a fixed priority scheduling (FPS), say rate monotonic scheduling. The component is schedulable under FPS if and only if $I_{FPS} \preceq I_{C_1} \parallel \ldots \parallel I_{C_n}$. How to capture the scheduling of a task set under FPS in terms of an interface I_{FPS} is described in [6]. A segregation property $B_{I_{FPS}}$ of interface I_{FPS} characterizes the resource demands of $C = \langle\{C_1, \ldots, C_n\}, FPS\rangle$.

The resource demands of a component C, explicated by the demand bound function $dbf_C(\Delta)$ can be safely over-approximated by any function $f(\Delta)$, with $f(\Delta) \geq dbf_C(\Delta)$. For example in [28] a linear function $ldbf_C(\Delta)$ is given for EDF scheduling that provides an upper bound for $dbf_C(\Delta)$. In our framework over-approximations of the resource demands B_{I_C} of a component translate into abstractions of B_{I_C}. As discussed before refinement of segregation properties (see Definition 17) is defined in a way such that every $B_{I_C} \preceq B'_{I_C}$ is also a segregation property for interface I_C, albeit a potential over-approximation of the resource demands defined by B_{I_C}.

Resource Model and Schedulability: Consider an explicit deadline periodic resource model $\Omega = (\Pi, \Theta, \Delta)$. It characterizes a partitioned resource that repetitively supplies Θ units of resource within Δ time units, with Π the period of repetition. The partitioned resource characterized by Ω, can also be characterized by the following slot reservation:

$$B_\Omega = 0^{\leq(\Pi-\Theta)}[(1^\Theta \,|||\, 0^{\Delta-\Theta})0^{\Pi-\Delta}]^\omega$$

The *resource supply* of a resource is the amount of provided resource allocations. Complementary to the demand bound function for components, the resource supply bound function $sfb_\Omega(\Delta)$ computes the minimum resource supply for Ω in a given time interval of length Δ.

The resource supply $sfb_\Omega(\Delta)$ can be safely under-approximated by any function $f(\Delta)$, with $f(\Delta) \leq sfb_\Omega(\Delta)$. Analogously, in our framework under-approximations of the resource supply are captured by the refinement relation. B'_Ω, with $B'_\Omega \preceq B_\Omega$, is a potential under-approximation of the resource supply of a resource as illustrated by

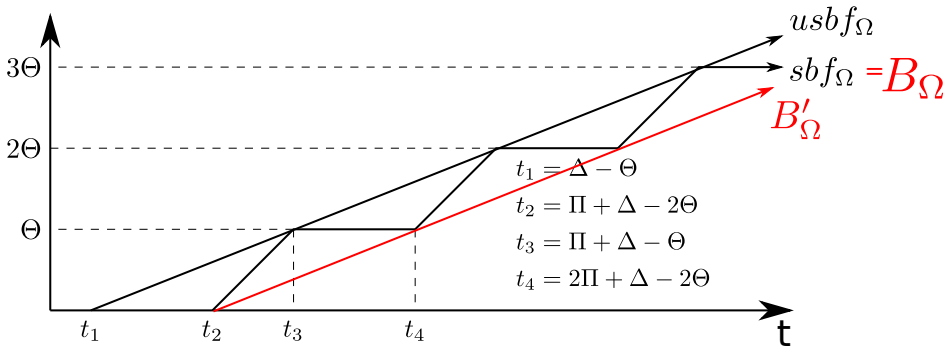


Figure 2.9: $sfb_\Omega(\Delta)$ of EDP model $\Omega = (\Pi, \Theta, \Delta)$

Figure 2.9.

In the context of EDP resource models, schedulability is defined for a real-time component $C = \langle \{C_1, \ldots, C_n\}, S \rangle$ using an EDP resource model Ω. Exact schedulability conditions are given for the scheduling strategies RM, DM and EDF. We will not go into the details of the theorems here and refer to [10] instead. Basically it must holds that $\forall \Delta : dfb_C(\Delta) \leq sfb_\Omega(\Delta)$. Schedulability of C under Ω can be formulated in our framework as refinement. Given the segregation property B_{I_C} and the resource supply B_Ω, then C is schedulable under Ω, if $B_{I_C} \preceq B_\Omega$. Sufficient conditions based on over-approximation of resource demands and under-approximation of resource supplies are induced by transitivity of the refinement relation. Given segregation property B_{I_C}, slot reservation B_Ω and $B_{I_C} \preceq B'_{I_C}$, and $B'_\Omega \preceq B_\Omega$, then $B'_{I_C} \preceq B'_\Omega \implies B_{I_C} \preceq B_\Omega$.

2.2.6 From Real-time model to real-time interfaces

In this section we present 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 [27], 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 (see Definition 4) and deadlines. As interfaces essentially are ω-languages, finite state machines (FSM) is 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. In advance it is checked whether this behaviour satisfies all given timing requirements, which are given in terms of contracts. If this is given, 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.

2.2.6.1 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 [19, 20] 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 $P_{ini} \cap P_{inj} =$

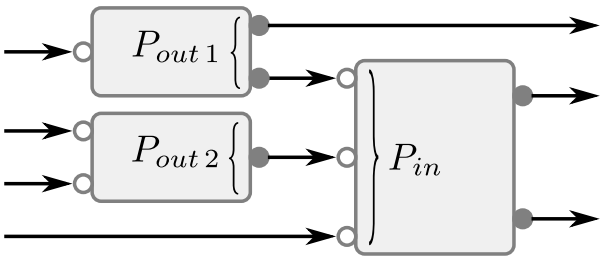


Figure 2.10: Port Composition

$P_{out\,i} \cap P_{out\,j} = \emptyset$ holds. Composition of $\mathcal{S}$ is the FST $F_{\mathcal{S}} = (\Sigma_P, P_{in}, P_{out}, S, s_0, T, G)$ where

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

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. The composed output hence 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 $\sigma_j|_{\epsilon P_{out}} = G(s_1, \ldots, s_m)|_{\epsilon P_{in\,j}}$. 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 2.10 depicts the relationship.

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* $G(s_1, \ldots, s_m)|_{\epsilon P_{ini}}$.
   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.

2.2.6.2 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

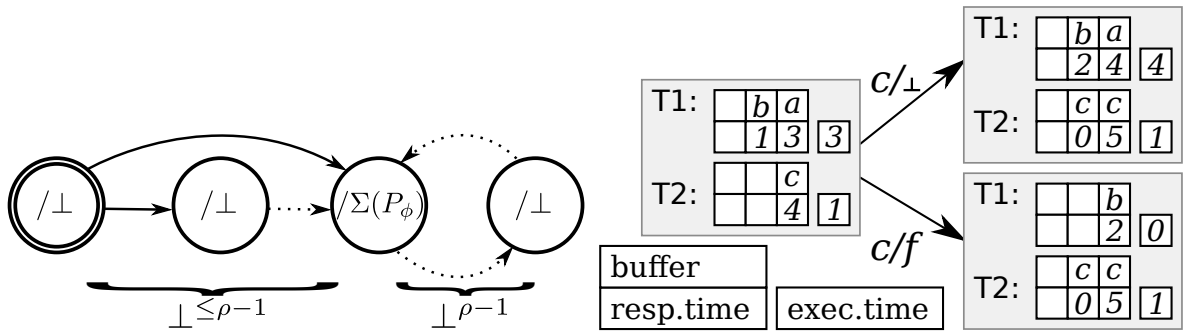
- S such that $|S| = |\mathcal{D}|$, and $\nu : S \rightarrow \mathcal{D}$ is a bijective mapping where $\nu(s_0) = v_0$,
- $(s, \sigma, s') \in T \iff \nu(s') \in m(\nu(s), \sigma)$,
- $G(s) = \sigma \iff g(\nu(s)) = \sigma$.

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.

2.2.6.3 State Space Construction

Definition 18 *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.* ◇



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

Figure 2.11: State representation for event sources and resources

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 straight forward. It is in fact a discretised version of the automaton templates defined by Hendriks and Verhoef in [15]. The FST constructed for a simple periodic event source $\phi = (P_\phi, \rho, \rho, 0)$ is depicted in Figure 2.11a. 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.

Figure 2.11b 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 2.11b, 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 2.11b this is for task τ_1 as it has higher priority. Calculation of successor states includes the decision whether tasks are completed. In Figure 2.11b, 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.

3 Integrated Framework

In this chapter we will describe a mapping from OSSS modeling primitives to real-time models as per Definition 4. The approach outlined in Section 2.2.6 can be used to obtain a real-time interface for a given real-time model. This allows us to apply the concept of resource segregation (see Section 2.2.5) for these real-time interfaces. We will discuss how segregation properties can be used from a methodological point of view when deploying OSSS application layer models on an OSSS virtual target architecture. Thereby, we define an integrated methodology/framework how deployment of different OSSS applications on the same virtual target architecture can be analyzed in a compositional way.

3.1 Mapping of OSSS to Analysis Model

3.1.1 Application layer

In Section 2.1.1 the modeling concepts of the OSSS application layer are introduced. The two main modeling primitives *Task* and *Shared Object* will each be mapped to a task $\tau \in \mathcal{T}$. Given an OSSS task $\mathsf{T} = (\mathsf{P}, \pi, \mathsf{d}, \Gamma)$, we assume that:

- It has exactly one activation port
- The CDFG of the implementation of the task can be simplified such that calls to shared objects (task communication) only take place at the beginning and the end of the task.

These assumptions allow us to interpret the different values observable at the activation port of T in an abstract manner, representing them by means of different events $\Sigma(p_\tau^I)$ observable at the input port of p_τ^I of the task τ. We can then determine an execution time interval for the range of values represented by each $\sigma \in \Sigma(p_\tau^I)$ (by simulation or static analysis). These execution time intervals denote the time from activation of T until a particular service is requested on one of its ports. All of these ports of T then correspond each to an output port $p_o \in P_\tau^O$ of τ and the different services requested on each of these ports coincide with the events $\Sigma(p_o)$ observable at that port. For an OSSS task T whose activation port is not connected, we create an event source $\phi \in \Phi$ with exactly one output port connected to the input port of the task τ corresponding to T. ρ^- and ρ^+ of ϕ are choosen equal to the period of the OSSS task, i.e. $\rho^- = \rho^+ = \pi$.

Given an OSSS shared object $\mathsf{SO} = (\mathcal{S}, \mathcal{I}, \Delta)$ we represent each of its services $\mathcal{S}$ by an event observable at the input port p_τ^I of the task τ. Further for each service we add an output port $p_o \in P_\tau^O$ to the task τ, each with a singleton event set $\Sigma(p_o)$. The different

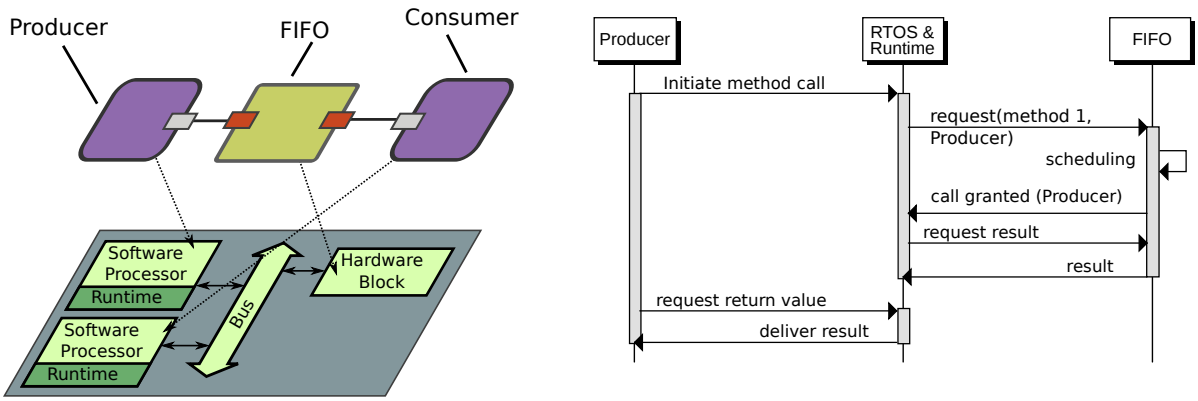


Figure 3.1: Simple Producer/Consumer OSSS model and shared object access protocol

execution time intervals, assigned by Δ to each service from the set of service $\mathcal{S}$ provided by SO, the become the execution time interval of an output specification $\psi \in \Psi$. The map Γ of task τ is defined accordingly.

In summary, we obtain a task network $TN = (\Sigma, P, \Phi, \mathcal{T})$ for an OSSS application layer model, with a 1:1 correspondence between each $\tau \in \mathcal{T}$ and an OSSS task T or an OSSS shared object SO. Further TN will contain event sources for OSSS tasks with open input ports.

As an example, consider the simple OSSS application model shown in Figure 3.1 (top left). The corresponding task network TN would consist of three tasks, one for each of the application elements, and one event source connected to the task representing the OSSS task *Producer*.

3.1.2 Virtual Target Architecture Layer

While one might be tempted to simply consider all elements of an OSSS virtual target architecture as resources $r \in R$ of a real-time model $\mathcal{A} = (TN, R, \Xi)$, this mapping is not quite correct. The arising issue becomes clear even in the very simple example shown in Figure 3.1. The application model consists of two OSSS tasks *Producer* and *Consumer*. As their names suggest, the task *Producer* carries out some computation and stores the results in a shared object *FIFO*, whose modeled functionality follows semantics of a typical FIFO. Finally, the tokens stored in *FIFO* are read by the task *Consumer*.

In the application model calling the service `put()` of *FIFO* from the task *Producer*, might look as simple as:

```
result = my_shared->put();
```

However, as shown on the right hand side of Figure 3.1, in order to implement this method call on the given target architecture, additional runtime support has to be generated. This consists of advancing the RMI protocol state according to it's phases covered in Section 2.1.3 and communicating any relevant method requests or state changes to the Shared Object scheduler. Any temporal effects of this protocol implementation have to

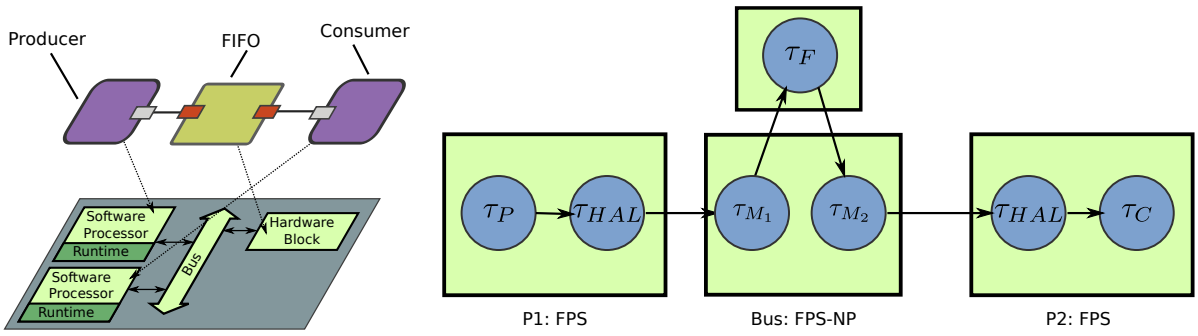


Figure 3.2: Simple Producer/Consumer OSSS model and task network

be captured in the real-time model. A typical approach is to require that such effects are subsumed by the execution time interval of the calling task. Here, we take a different approach, namely making these protocol overheads explicit by means of an additional task τ_{HAL}. We model such a task τ_{HAL} for every element of the virtual target architecture that is a master on the SoC bus. Consequently, every OSSS task T accessing a remote shared object, i.e. not located on the same processor, does so by letting τ_{HAL} perform the requested service. This approach comes with the benefit that we can explore different deployment scenarios for the shared objects of the application model without having to change the OSSS tasks in terms of their execution time annotations (cf. Section 2.1.4) or the execution time intervals of the output specifications Ψ of the corresponding task τ.

While this approach considers the temporal effects incurred by the protocol on each processor, the communication with a remote shared object must traverse the SoC bus or another communication link, like a point-to-point channel. Obviously, also this communication between two different hardware elements via a bus takes time. Here we adopt the typical strategy to represent such a communication by a task τ and the communication link by a resource. Especially, the SoC bus is interesting here, because it is a resource shared by all bus masters connected to it. Hence, access to the bus must be arbitrated, or put in terms of the real-time model: The bus is a resource that is scheduled.

In summary, given an OSSS virtual target architecture and an OSSS application model mapped to the architecture, we obtain a real-time model $\mathcal{A} = (TN, R, \Xi)$, where there is a 1:1 correspondence between its resources $r \in R$ and the hardware components of the virtual target architecture. In addition the task network TN is different from the task network obtained when only considering the application model, as it further contains a task for each message traversing the SoC bus and an additional task per master of the SoC bus.

Considering again the simple producer/consumer example, the corresponding real-time model is illustrated in Figure 3.2 (right).

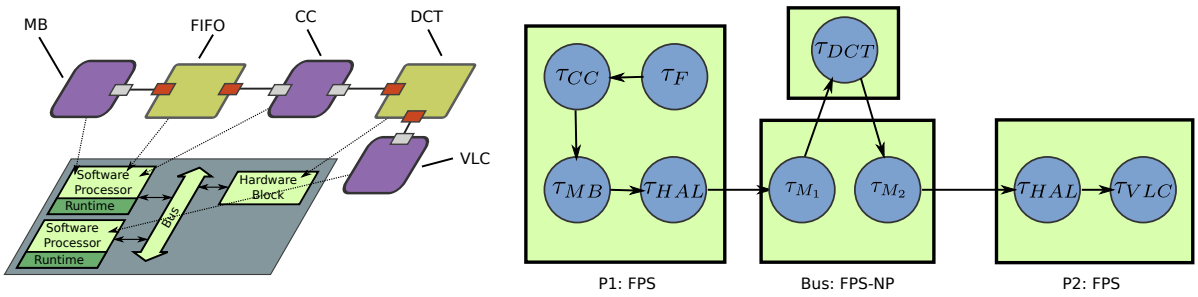


Figure 3.3: OSSS model of a JPEG encoder and corresponding task network

3.2 Resource Segregation

Having defined a mapping of OSSS application models and virtual target architecture models to a real-time model, the approach outlined in Section 2.2.6 can be used to obtain a real-time interface for a given real-time model, representing all possible behavior of the OSSS application model with respect to the underlying virtual target architecture. Thus, we can formally analyze the OSSS model and e.g. check satisfaction of safety properties.

In this section we will describe how the concept of segregation properties can be applied to capture a set of time-partitions of the resources of a virtual multi-core target architecture. This enables a compositional deployment analysis of different OSSS applications hosted on the same architecture in terms of their respective resource segregation properties. First, reconsider the basic strategy how we intend to make use of the segregation property concept and focus on Figure 2.8 (see page 26). We want to define a slot reservation language of the components of a virtual target architecture for each application. Assuming we have two applications, we define segregation properties $B_{part_1} = B_{R_{1_1}} \parallel \ldots \parallel B_{R_{1_n}}$ and $B_{part_2} = B_{R_{2_1}} \parallel \ldots \parallel B_{R_{2_n}}$. Obviously, we have to be sure, that we can implement B_{part_1} and B_{part_2} on the target architecture, i.e. $B_{part_1} \parallel B_{part_2} \neq \emptyset$. According to Definition 15, we must prove that B_{part_i} is a segregation property for the real-time interface obtained from the real-time model corresponding to the respective application.

While at a first glance the concept of segregation properties directly applies to OSSS models and their implementations, we identified two issues: The first issue is about to share an OSSS shared object not only between the tasks of the same application, but also between different applications. As an example consider a JPEG encoder, whose OSSS model and the corresponding task network are shown in Figure 3.3. The JPEG encoder consists of four task. The task *MB* samples macroblocks parsed from a given BMP file. Each macroblock is send via a fifo to the task *CC* implementing a color conversion. The task *CC* in turn sends its processed macroblock to a discrete consine transform (DCT) modeled as an OSSS shared object. The *DCT* functionality is implemented as a dedicated hardware resource. Once transformed, the output is picked up by a task *VLC*, implementing a variable length coding. Obviously, the designer could be interested to share the DCT functionality and also make use of it within the context of another application deployed on the very same target architecture. When talking about software,

resource segregation can be easily implemented, e.g. by means of a hierarchical scheduling scheme. The higher level scheduler would then switch from one application context to another application context, just as required by the respective segregation property. Depending on the alignment of the segregation properties and activation patterns of the tasks of the applications, tasks may be preempted when switching application contexts. While this perfectly works in the software domain, this will not work when sharing the functionality provided by dedicated hardware elements. Typical implementations in dedicated hardware simply do not support such kind of preemption. Hence, tasks representing shared objects are always non-preemptive. The second issue is related and stems from an implementation requirement that also the previously introduced tasks τ_{HAL}, implementing the RMI protocol state machine when communicating with a remote shared object, must not be preempted.

These issues do not rule out the concept of segregation properties to be applied, but it is important to observe that we need specialized patterns to specify slot reservations, where there are multiple consecutive slots reserved such that non-preemptable tasks can execute. Consequently, we cannot use for example the resource models proposed by I. Lee et. al., which we have briefly introduced in Section 2.2.5.1. This becomes clear when reconsidering the slot reservation characterized by an EDP model $\Omega = (\Pi, \Theta, \Delta)$:

$$B_\Omega = 0^{\leq(\Pi-\Theta)}[(1^\Theta \,|||\, 0^{\Delta-\Theta})0^{\Pi-\Delta}]^\omega$$

As we can see the slot reservation characterizes an arbitrary distribution of Θ slots over a time length of Δ that repeats with a period of Π. Definition 15 requires that an interface is schedulable under *all* slot reservations defined by B_Ω, for B_Ω to be a segregation property for the interface. However, the legal schedules $L_K|_R$ of the interface I_K, will only contain those schedules where e.g. τ_{HAL} is not preempted. That means every Ω with $\Theta \neq \Delta$ cannot be a segregation property for an interface containg non-preemptable tasks. Note, that only allowing resource models with $\Theta = \Delta$ does not "fix" the issue, due to the non-deterministic offset $0^{\leq(\Pi-\Theta)}$ of B_Ω. Precisely, this offset renders a non-deterministic choice about when the first reserved slot occurs. A similar effect can be observed for typical activation patterns of tasks. Consider the real-time interface described in Example 1 (see page 16) of a simple periodic task with period $p = 5$ and execution time $c = 3$. The language $L_K|_R = 0^{<5}[\tau^3 \,|||\, 0^2]^\omega$ characterizing the legal schedules also exhibits such a non-deterministic offset conforming to the well known formalism of event streams. This offset renders a non-deterministic choice about all possible phasings of the initial activation of the task. Due to these phasings of task activations and EDP resource models, we cannot guarantee that a non-preemptable task gets an uninterrupted sequence of reserved slots at the time it gets activated.

A possible solution for obtaining a segregation property for a non-preemptive task is to eliminate both non-determinisms. Considering a simple non-preemptive task with period $p = 5$ and execution time $c = 3$, its slot language be then be $L_K|_R = [\tau^3 0^2]^\omega$. A segregation property for this non-preemptable task, is then identical to L_K. This way we can ensure that the task always has 3 slots to execute. For the remaining tasks, segregation properties can still be specified using for example EDP resource models.

3.3 Segregation in OSSS executable specifications

Until now, OSSS has not been designed to integrate multiple different applications in a single system, although on the Application Layer, distinct connected components in the task/Shared Object graph can be seen as distinct applications already. In such a case, no (explicit) resource sharing occurs between such partitions.

In order to address the segregation problem on the Virtual Target Architecture, an application as a set of tasks and Shared Objects needs to be introduced explicitly in the OSSS application layer model. Secondly, resource budgets (CPU processing time, memory requirements, bus slots, other platform resources) need to be assigned, for instance in terms of abstract processing elements.

Afterwards, the mapping of the application layer elements (tasks and Shared Objects) are mapped to the "real" processing elements on the technical architecture of the system. In this section, we will first elaborate on hierarchical scheduling for task sets of different applications. Then, we extend this idea in order to support resource segregation properties on non-preemptive shared resources.

3.3.1 Multiple applications and hierarchical scheduling

A straight-forward approach for combining task sets of several applications on a common processing unit is the introduction of a second scheduling hierarchy. This new *root scheduler*, as can be seen in Figure 3.4, manages resource usages according to some budget attached to each application-specific scheduler. A key challenge regarding embedded systems is the ability of handling the additional complexity introduced by the new scheduling level in a predictable manner. Timing constraints and existing assumptions about worst case execution times have to be carefully revisited and analyzed.

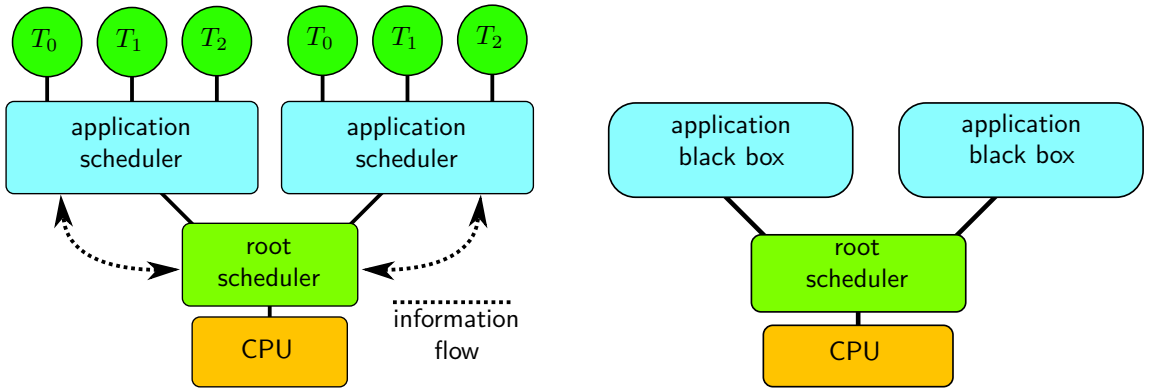


Figure 3.4: Dependent and independent hierarchical scheduling of different applications.

In [17], several approaches from the state-of-the-art on virtualization techniques for combining task networks on processing units were presented and classified. Especially the past and recent work on multi-level hierarchical scheduling with both independent and dependent task sets was discussed.

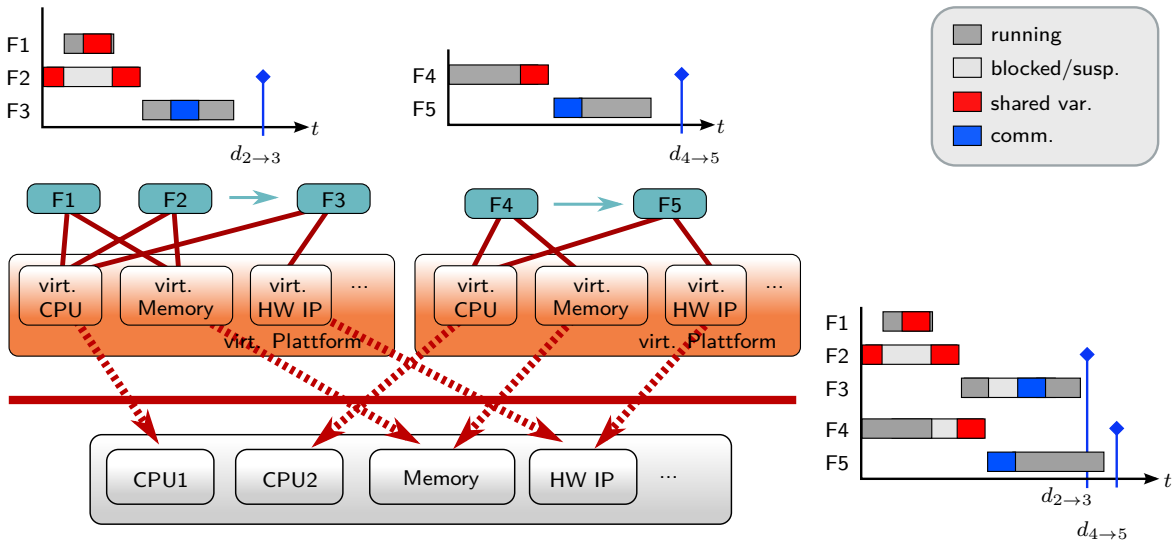


Figure 3.5: Combining applications using virtualization.

One property that allows us to classify hierarchical scheduling approaches is the level of independence between *root-* and *application-level* scheduling algorithms, as illustrated in Figure 3.4. In an independent setup of these levels, the schedulers do not cooperate, i. e. they do not share any information about the state of their tasks. In a dependent setup, decisions of a scheduling policy from a scheduler of one level are not only based on the state of its direct siblings (applications or tasks) but also on its indirect siblings. In a dependent hierarchical scheduling setup, the root scheduler is therefore able to perform decisions based on the task states of each application. Independence between scheduling levels increases modularity but may also decrease the overall performance capabilities or even the schedulability in terms of real-time constraints, since root schedulers cannot take the whole system state into account.

A future OSSS extension shall support these different types of hierarchical scheduling by providing a combined abstraction for implementing and analyzing scheduling policies for processing elements with budgets attached to them as well as the mapped "virtual" processing units, where the final task execution order is determined.

3.3.2 Segregation properties for resources

If we extend hierarchical scheduling and its implementation in terms of virtualization to also include other platform resources, we can effectively provide a virtualized environment for each application and analyze any impacts of integrating them, as illustrated in Figure 3.5. In this section, we will propose an approach for modeling resource sharing across applications. As we have already discussed in Section 3.2, when combining several applications, the designer might want to re-use not only processing elements for task allocation between applications, but also shared objects.

To support this mapping step, we introduce a new modelling element called *passive*

resource unit, which represents a shared resource of the system. Much like the relation between tasks and processing units, the passive resource unit is a platform component which can be used to map entities of the application layer on the platform. However, instead of executing tasks, passive resource units are meant to execute requests to mapped hardware shared objects. Consider again Figure 3.3 on page 39, and the intention of the designer to share the DCT functionality between different applications. In order to simulate and analyze such a decision, OSSS has to provide a passive resource unit which handles any context changes between the mapped object entities on the shared resource. The passive resource unit therefore has to manage the state of each mapped shared object by saving and restoring it when the application context switches.

The *shared object state* will therefore still be unique to each application, whereas the *functionality* can be shared across applications. Incoming requests are scheduled by the passive resource unit and forwarded along with the application-specific shared object state in order to apply the actual method behavior on the correct state. This of course introduces a new scheduling hierarchy level in the passive resource unit, similar to the hierarchical scheduling of tasks as presented in Section 3.3.1. But as we have discussed in Section 3.2, the key problem is to align requests for the execution of non-preemptive resources from different applications with their corresponding task activation patterns. Our simulation framework should therefore be able to expose such behavior and enable the designer to further analyze resource usage contention across applications.

The needed scheduling hierarchy is implemented by providing a root scheduler attached to the passive resource unit, scheduling between requests coming from different applications. Any requests initiated by tasks of an application must be associated to the originating application when they are processed by the passive resource unit. It is then possible to define scheduling policies which schedule requests according to budgets defined for each application, i. e. on a percentage basis.

As already explained in Section 3.3.1, hierarchical scheduling policies mainly differ in their amount of information exchange between each hierarchy. This is also the case for passive resource units. A shared object state can be interpreted as a black box for the passive resource unit, or one could define an interface for exchanging information between these levels to schedule requests more effectively. Again, our framework extension should provide a combined abstraction for exploring the impact of these scheduling implementations on the system performance and schedulability in terms of the given resource segregation properties.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publ., Boston, MA, USA, 1986.

[2] The ATESST Consortium. *EAST ADL 2.0 Specification*, February 2008.

[3] AUTOSAR GbR. *Software Component Template*, November 2009. Version 4.0.0.

[4] SanjoyK. Baruah, LouisE. Rosier, and RodneyR. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.

[5] 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.

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

[7] Alan Burns and Andy Wellings. *Concurrency in Ada*. Cambridge University Press, 1997.

[8] S. Chakraborty, S. Kunzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2003*, pages 190–195. IEEE Computer Society, 2003.

[9] Werner Damm, Hardi Hungar, Stefan Henkler, Thomas Peikenkamp, Ingo Stierand, Bernhard Josko, Philipp Reinkemeier, Andreas Baumgart, Matthias Büker, Tayfun Gezgin, Günter Ehmen, Raphael Weber, Markus Oertel, and Eckard Böde. Architecture modeling. Technical report, OFFIS, March 2011. `http://www.offis.de/publikationen/detail/architecture-modeling.html`.

[10] Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using edp resource models. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 129–138. IEEE Computer Society, 2007.

[11] Fossy – *Functional Oldenburg System Synthesiser*. `http://fossy.offis.de`.

[12] K. Grüttner, C. Grabbe, F. Oppenheimer, and W. Nebel. Object Oriented Design and Synthesis of Communication in HW/SW Systems with OSSS. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), Hokkaido, Japan*, October 2007.

[13] Kim Grüttner, Herrholz Andreas, Philipp A. Hartmann, Andreas Schallenberg, and Claus Brunzema. *OSSS - A Library for Synthesisable System Level Models in SystemC*™, 2008. `http://www.system-synthesis.org`.

[14] Philipp Andreas Hartmann, Philipp Reinkemeier, Henning Kleen, and Wolfgang Nebel. Efficient modelling and simulation of embedded software multi-tasking using SystemC and OSSS. In *Forum on Specification, Verification and Design Languages, 2008. FDL 2008.*, pages 19–24, September 2008.

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

[16] T.A. Henzinger and S. Matic. An interface algebra for real-time components. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pages 253–266. IEEE Computer Society, 2006.

[17] Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner, and Achim Rettberg. Hierarchical Real-Time Scheduling in the Multi-Core Era – An Overview. In *Fourth IEEE Workshop on Self-Organizing Real-Time Systems (SORT'2013)*, Paderborn, Germany, June 2013.

[18] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society, 1989.

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

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

[21] Object Management Group. *OMG Systems Modeling Language (OMG SysML ™)*, november 2008. Version 1.1.

[22] Philipp Reinkemeier and Ingo Stierand. Compositional timing analysis of real-time systems based on resource segregation abstraction. In Gunar Schirner, Marcelo Götz, Achim Rettberg, MauroC. Zanella, and FranzJ. Rammig, editors, *Embedded Systems: Design, Analysis and Verification*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 181–192. Springer Berlin Heidelberg, 2013.

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

[24] Alberto Sangiovanni-Vincentelli. Is a unified methodology for system-level design possible? *IEEE Design and Test of Computers*, 25(4):346–357, 2008.

[25] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, 18(6):23–33, 2001.

[26] Gunar Schirner and Rainer Dömer. Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling. In *Proc. of Design, Automation and Test in Europe (DATE'2008)*, pages 122–127, Munich, Germany, March 2008.

[27] 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.

[28] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, RTSS '03, pages 2–13. IEEE Computer Society, 2003.

[29] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. Fast and Accurate Resource Conflict Simulation for Performance Analysis of Multi-Core Systems. In *Proc. of Design, Automation and Test in Europe (DATE'2011)*, pages 210–216, Grenoble, France, March 2011.

[30] Ingo Stierand, Philipp Reinkemeier, Tayfun Gezgin, and Purandar Bhaduri. Real-time scheduling interfaces and contracts for the design of distributed embedded systems. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, 2013.

[31] Lothar Thiele, Ernesto Wandeler, and Nikolay Stoimenov. Real-time interfaces for composing real-time systems. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, EMSOFT '06, pages 34–43. ACM, 2006.

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

[33] Ernesto Wandeler and Lothar Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pages 243–252. IEEE Computer Society, 2006.

[34] Henning Zabel and W. Müller. An Efficient Time Annotation Technique in Abstract RTOS Simulations for Multiprocessor Task Migration. In *IFIP, TC10 Working Conf. on Distributed and Parallel Embedded Systems (DIPES'2008)*, volume 271 of *IFIP*, pages 181–190, Milano, Italy, September 2008.