Model Composition for Scheduling Analysis in Platform Design

Kai Richter, Dirk Ziegenbein, Marek Jersak, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
D-38106 Braunschweig / Germany
{richter|ziegenbein|jersak|ernst}@ida.ing.tu-bs.de

Categories and Subject Descriptors

C.0 [**General**]: System Architecture; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*; C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Algorithms, Performance, Verification

Keywords

Platform-Based Design, Performance Analysis, Scheduling, Formal Analysis

Abstract

We present a compositional approach to analyze timing behavior of complex platforms with different scheduling strategies. The approach uses event interfacing in order to couple previously incompatible analysis techniques which provide subsystem and component behavior. Based on these interfaces, event propagation using abstract models is used to derive global system timing properties.

1. INTRODUCTION

Embedded system platforms consist of a combination of different processor types, specialized memories, and weakly programmable or fixed function components with a communication infrastructure consisting of busses, switches or point-to-point connections. Programmability and configurable software architectures are key to adapt platforms to a variety of applications. The verification of such platforms faces two main problems, verification of the system function and verification of the platform implementation, i.e. the adherence to system timing constraints, memory requirements, or power consumption. In this paper, we will focus on timing verification.

In traditional hardware design, system timing can usually be derived by hierarchical composition of individual component timing. This is possible since in most cases, component control is single threaded following a fixed control sequence which only depends on input patterns. Optimization is concerned with an optimal static schedule of operations. Behavioral synthesis closely follows this approach [7].

Embedded software adds process preemption to enable another class of scheduling strategies. Preemptive and time-driven scheduling introduces timing dependencies between functionally independent processes.

Heterogeneous platforms take the next step of target system complexity combining several preemptive and non-preemptive scheduling strategies in one system, e. g. a static schedule on a DSP and a priority-driven schedule on a microcontroller. Communication adds to behavioral complexity by introducing additional resource sharing strategies. Buffering changes the memory requirements and can lead to internal event bursts.

The analysis of such a platform is a major challenge and is currently limited to simulation approaches, such as in VCC [1], or Seamless [11]. The known limitations of simulation such as incomplete coverage and corner case identification are aggravated since many of the design errors only result from system integration requiring detailed knowledge which is often not available to the integrator. As a result, simulation does not provide conservative system properties, in particular timing. In case certain timing properties have to be guaranteed, static formal analysis is the only option.

Target system timing analysis can be divided in two parts: analysis of a single process executed on a target system component (i. e. the core execution time), and analysis of resource sharing effects on the response time of a process based on given core execution times. For the first part, there are many recent contributions combining implicit or explicit program path analysis and cycle-true processor modeling, such as [9, 3, 17, 5]. In addition, process communication can be determined to analyze communication channel load and timing [17]. For the second part, there is a huge amount of work, mainly in the domain of real-time operating systems [10, 6]. These approaches capture system timing using equations and provide appropriate solution algorithms. Unfortunately, a single coherent scheduling strategy for a given system, whether single or multiprocessor, is assumed in literature. There are very few exceptions which consider special cases of more complex architectures, such as [13] analyzing response times for static-priority process scheduling combined with a TDMA bus protocol. However, it is doubtful that a general approach to a self-contained solution for arbitrarily complex platform architectures can be found, mainly because of the highly complex dependencies in such systems.

The approach presented in this paper is based on analysis coupling rather than finding self-contained solutions. We identify architecture components for which an appropriate analysis exists. Then, we combine these analysis approaches in order to obtain a compositional description of the complex system-level timing behavior. This is not an easy task, mainly because of the incompatibilities of the used event models. Most of the analysis approaches in literature assume certain input event models that lead to process or communication execution. In effect, the input events determine the system workload. In distributed platforms, the input events of one component result from the output events of the connected components. For our compositional approach, it is required that the output event models of one architecture component are compatible with the input event models of the connected components. Otherwise, they can not be reasonably coupled. Interestingly, the importance of output event models has been widely neglected. In this paper, we will first present ways to derive output event models from the existing analysis approaches. We will see that these output event models are usually different from the input event models. To overcome the problem of incompatible models at inputs and outputs, we use transformation functions to adapt event models. Finally, we show the actual composition of analysis approaches. We introduce the concept of analysis domains as the key entities for the composition process. These analysis domains are represented by abstract timing characteristics rather than detailed behavior.

DAC 2002, June 10-14, 2002, New Orleans, Louisiana, USA.

The paper is organized as follows. The next section reviews existing work in the area of real-time scheduling analysis and event models. Section 3 contains some preliminaries: the derivation of output event models and further analysis assumptions. In Section 4, we show how analyzable subsystems (analysis domains) can be identified and represented. The actual composition is explained in Section 5. After an example of a complex analysis problem in Section 6, we conclude the paper with an outlook on future work.

2. STATE OF THE ART

2.1 Analyzing Scheduling Strategies

In the area of real-time operating systems, there are several substantially different resource sharing (scheduling) strategies. Preemptive scheduling based on static priorities (e. g. rate-monotonic), dynamic priorities (e. g. earliest deadline first), and time-slicing (e. g. time division multiple access and round robin) are among the most important strategies. For each of the mentioned strategies, a set of analysis approaches exist. The approaches take the scheduling strategy and the core execution times as input to a self-contained mathematical description in order to calculate conservative bounds on the response times of processes.

In the early 70s, Liu and Layland proposed a preemptive priority-driven scheduling to guarantee deadlines for periodic hard real-time processes [10]. They considered a static (Rate Monotonic) and a dynamic (Earliest Deadline First) priority assignment and provided a formal analysis framework for both. In [6], Kopetz and Gruensteindl proposed TTP (time triggered protocol) for communication scheduling in distributed systems and presented an analysis. TTP implements the TDMA (time division multiple access) scheduling strategy. Both contributions assume a periodic activation of processes. Recent extensions of the mentioned work allow periodic activation with jitter, e. g. [15], and arbitrary deadlines and burst [8] for static-priority scheduling. Sprunt et. al. [16] analyze the influence of sporadic process activation. Unfortunately, all mentioned approaches assume a single coherent scheduling strategy for a given system, whether single or multiprocessor. Very few exceptions consider special cases of more complex architectures, such as [13] analyzing response times for static priority process scheduling combined with a TDMA bus protocol.

So far, there is no general approach to a self-contained solution for arbitrarily complex systems. In contrast, we propose a compositional approach. Currently, this is not possible due to incompatible assumptions on the type of process activation, e. g. periodic, jitter, burst. In the existing analysis approaches, the activation of processes is usually captured using abstract event models.

2.2 Event Models

In the literature on real-time analysis, there are four event models of major importance. A simple and efficient assumption is a stream of *periodic* input events. Here, the arrival of events can be captured by a single parameter, the period T. Often, periodic events are allowed to deviate with respect to their period. This adds another parameter to the periodic model, the *jitter* (J). Other models capture *bursts* of events. A burst is characterized by a number of events (burst length b) within a given time interval (the outer period T). This outer period may also jitter (J). If known, a minimum time distance (the inter-arrival time t) between two successive events within a burst can be specified. *Sporadic* events are captured by the minimum inter-arrival time t, only. Gresser [4] provides a more general model. It introduces vectors of sequential time intervals. However, the model is not applicable to most of the analysis approaches.

In a recent publication [14], we have presented an approach to *interface* between different event models. Table 1 shows simple transformation functions, which we call event model interfaces (EMIFs), between these models. An EMIF transforms the representation of an event stream from one event model into the abstract parameters of another event model. For instance, a periodic event stream with the period T_X can be captured by the burst event model, when we set the burst length to $b_Y = 1$, the outer burst period to $T_Y = T_X$, and the minimum inter-arrival time to $t_Y = T_X$.

Note that such EMIFs do not modify the actual event streams, rather they transform the abstract representation of a single event stream. The transformations are uni-directional, and can not be found for all combinations of event models. For instance, an event stream with burst can not be captured by the parameters of a purely periodic event model. In this case, we need to adapt the event stream itself. This can be done by adding functionality to the system, the so called event adaptation functions (EAF). A periodic event stream with jitter can be re-synchronized by means of a buffer with a periodic output issue rate. The same applies to the re-synchronization of event streams with burst. For detailed information about EMIFs and EAFs and the corresponding formalisms, we refer to [14]. There, we also derive worst-case buffer sizes for the EAFs and additional event delays which result from re-synchronization.

3. PRELIMINARIES

For our proposed composition of existing analysis approaches, the output event models of one architecture component have to be compatible to the assumed input event models of the connected components. However, output event models have been widely neglected in literature, and the analysis approaches do not characterize them by themselves. Instead of going into the details of the analysis approaches, we rather use their abstract characteristics (input event models and response times) to derive the output event models.

3.1 Output Event Models

The basic idea of deriving output event models is simply based on abstract event propagation through architecture components. An input event activates a dedicated function inside the component. A corresponding output event will occur after the function is completed, i. e. after the corresponding response time. When having a constant response time, each event experiences the same propagation delay. Thus, the output model is identical to the input model. However, in complex software systems only upper and lower bounds for the response time will be given, e. g. due to data dependent process execution times or a varying number of preemptions by other processes. For a periodic input model, the output is not purely periodic anymore, but will experience a jitter that equals the difference between the upper (t^+_{resp}) and the lower (t^-_{resp})response time bound. If the input events already arrive with a certain jitter, this jitter is additionally propagated to the output. In other words, both jitters –the internal and the external– are superposed:

$$J_{\text{out}} = J_{\text{in}} + \underbrace{t^+_{\text{resp}} - t^-_{\text{resp}}}_{\text{internal jitter}}$$

A more complicated situation can be found for input event models with burst. An overview of how output event models can be derived from the analysis characteristics (input event model and response time) is given in Table 2.

Input Event Model	Output Event Model
sporadic:	sporadic:
t_{in}	$t_{\text{out}} = \max\left(t^-_{\text{resp}}, t_{\text{in}} - (t^+_{\text{resp}} - t^-_{\text{resp}})\right)$
periodic:	periodic with jitter:
T_{in}	$T_{\text{out}} = T_{\text{in}}$, $J_{\text{out}} = t^+_{\text{resp}} - t^-_{\text{resp}}$
periodic with jitter:	periodic with jitter:
T_{in}, J_{in}	$T_{\text{out}} = T_{\text{in}}$
	$J_{\text{out}} = J_{\text{in}} + t^+_{\text{resp}} - t^-_{\text{resp}}$
periodic with burst:	periodic with burst and jitter:
T_{in}, b_{in}, t_{in}	$T_{\text{out}} = T_{\text{in}}$
	$b_{\text{out}} = b_{\text{in}}$
	$t_{\text{out}} = \max\left(t^-_{\text{resp}}, t - (t^+_{\text{resp}} - t^-_{\text{resp}})\right)$
	$J_{\text{out}} = t^+_{\text{resp}} - t^-_{\text{resp}}$

Table 2: Deriving Output Event Models

3.2 Input Event Interfaces

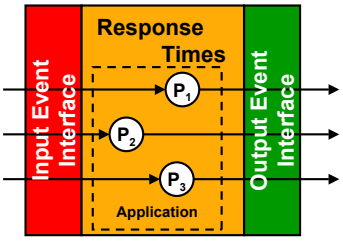
Above, we have shown how output event models can be derived. In our compositional approach, we will use the output event models of one component as the input event model of the connected components, possibly incorporating model transformations. In Section 2, we mentioned that the assumed input event models of the existing analysis approaches can be categorized into four classes. Additionally, some of the analysis approaches require a few more properties. For

$EMIF_{X \to Y}$	Y=periodic	Y=jitter	Y=burst	Y=sporadic
X=periodic	$T_Y = T_X$ (identity)	$T_Y = T_X, J_Y = 0$	$T_Y = T_X, b_Y = 1, t_Y = T_X$	$t_Y = T_X$
X=jitter	—	$T_Y = T_X, J_Y = J_X$ (identity)	—	$t_Y = T_X - J_X$
X=burst	—	—	$T_Y = T_X, b_Y = b_X, t_Y = t_X$ (identity)	$t_Y = t_X$
X=sporadic	—	—	—	$t_Y = t_X$ (identity)

Table 1: Event Model Interfaces for the Transformation of Event Models

example, the analysis of Liu and Layland [10] requires that the process response times do not exceed the corresponding activation periods. Lehoczky [8] provides an analysis assuming the same input event model (periodic) but without the additional constraint. In general, assuming a certain model is not enough to analyze scheduling. We will therefore use the term "input event interface" to describe an "input event model" together with additional assumptions. Correspondingly, we use the term "output event interface". We assume these "input event interfaces" to be provided by the analysis techniques employed.

4. ANALYSIS DOMAINS



Input Event Interface:	event model purely periodic($T_{P_i,in}$) additional assumption from analysis in [10]: $\forall P_i: t^+_{P_i,resp} < T_{P_i,in}$
Response Times:	`GetResponseMax(), GetResponseMin()` Example for `GetResponseMax()` for the analysis in [10]: `GetResponseMax()` = smallest positive roots of: $t^+_{P_i,resp} = t^+_{P_i,core} + \sum_{\forall P_j \in HP(P_i)} \left(\left\lceil \frac{t^+_{P_i,resp}}{T_{P_j,in}} \right\rceil \times t^+_{P_j,core} \right)$
Output Event Interface:	periodic ($T_{P_i,out}$) with jitter ($J_{P_i,out}$): $T_{P_i,out} = T_{P_i,in}$ $J_{P_i,out} = t^+_{P_i,resp} - t^-_{P_i,resp}$

Figure 1: Representation of Analysis Domain

As described in the introduction, we assume that a platform consists of disjoint parts that use different scheduling strategies. In the following, we will call these parts scheduling domains. In the design, these scheduling domains are coupled via input and output events between domains. We have introduced event interfaces to describe the interfacing between these domains. We assume that for each scheduling domain there is an analysis technique that allows to derive the timing of the domain. As described before, there is a host of solutions in this field which can directly be employed. This disjoint set of scheduling domains defines a corresponding set of analysis domains. More precisely, an analysis domain is an architecture component or subsystem for which an analysis technique is available. Figure 1 shows an analysis domain example.

As previously mentioned, the availability of analysis approaches for an architecture component strongly depends on the scheduling strategy of the component. If several analysis approaches exist, as it is the case for preemptive static-priority scheduling, one candidate has to be selected. The input event interface of the component is constrained by the analysis. If we can find an analysis for the given scheduling strategy, and if the input events are compatible with the assumed interface, the analysis allows us to calculate conservative response times of processes. From these, we can further derive the output event interface, as explained in Section 3.1. The process of analyzing architecture components and deriving analysis domains is depicted in Figure 2.

5. DOMAIN COUPLING

As previously mentioned, the well known scheduling analysis approaches assume certain input event interfaces. But as mentioned in Section 2.2, the output event interfaces do not necessarily directly fit any of the possible input interfaces of the connected analysis domains. In this section, we explain how such seemingly incompatible analysis domains can be coupled. We start by finding reasonable ways to *adapt* event models for comparatively simple systems without any feed-back in the process graph as well as in the analysis itself. Then, we will hierarchically compose compatible domains. Finally, we will investigate several types of feed-back within this analysis process.

5.1 Feed-Forward

We use an example, depicted in Figure 3, to demonstrate the coupling of analysis domains. In this example, the analysis domains have already been selected. The system consists of two analysis domains consisting of one processor each, CPU_1 and CPU_2. Either of the processors executes two processes. The processes P_1 and P_2 are activated by events coming from the systems environment. On completion, they activate the processes P_3 and P_4 which themselves produce output to the environment. From the specification, we know that process P_1 is activated periodically with a fixed period $T_{P_1,in} = 40$ms. Process P_2 is also activated periodically ($T_{P_2,in} = 20$ms) but with a jitter $J_{P_2,in} = 5$ms. The processes on CPU_1 are scheduled according to the rate-monotonic priority assignment [10], while a round robin scheduler alternately assigns CPU_2 to either of its processes. The time slots for P_3 and P_4 are $t_{P_3,slot} = 5$ms and $t_{P_4,slot} = 3$ms, respectively. The core execution time intervals of the processes are $[15,17]$ms for P_1, and $[8,11]$ms, $[10,11]$ms, and $[3,5]$ms for the processes P_2, P_3, and P_4, respectively. We are looking for conservative (upper and lower) bounds on the response times for each event that is input to the system until a corresponding event is output to the environment.

For simplicity, we would like to use the approach of Liu and Layland [10] to analyze the processes on CPU_1. For the processes on CPU_2, we want to use the analysis from [2]. An interesting point is that we do not even need to know about the details of the mentioned analysis approaches. We only need to know the restrictions on the input event interfaces and the algorithms to calculate the response times. In contrast to finding a dedicated self-contained analysis for the given problem (like e. g. [13] does), the event model interfaces (EMIFs) from [14] will help us to combine both analysis domains with only little additional effort.

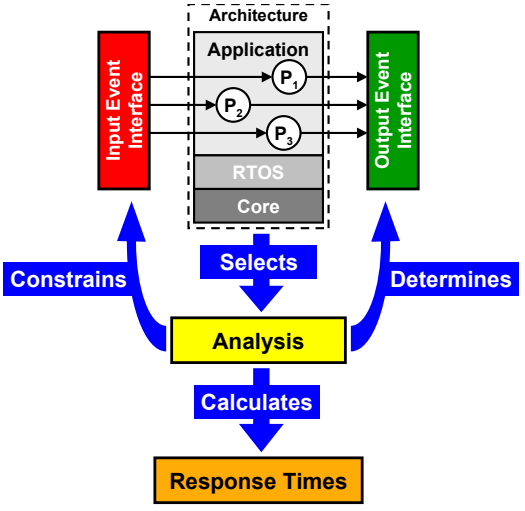


Figure 2: Local Analysis of Component

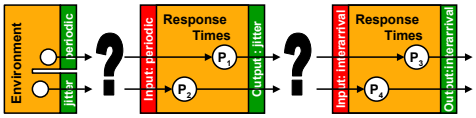


Figure 3: Coupling of Analysis Domains

In the example, the selected analysis approaches assume the input of CPU_1 to be purely periodic (this is not a limitation of the approach, just an example), while only minimum inter-arrival times are required to analyze the timing of CPU_2. Our first problem is the jitter on the input events for P_2. We can not use a simple model transformation, but have to transform the event stream itself by means of an appropriate event adaptation function. As explained in Section 2.2, we need an event buffer of size "one event" with an output issue rate of $T_{\mathsf{P}_2,\text{in}} = 20\text{ms}$ to adapt the input event stream to the required input interface of the analysis domain of CPU_1. By this, we re-synchronize the event stream to its period and eliminate the jitter. As can be seen in Figure 4, the EAF is represented the same way as the analysis domains. Although they have no real function, i. e. they just copy input events to output events, they have a dedicated timing behavior in terms of input and output event interfaces, and a response time, which in this case equals the jitter of the input event stream. Now, we can do the selected analysis. We obtain the response times:

$$\begin{aligned} t_{\mathsf{P}_1,\text{resp}} &= [23,39]\text{ms} \\ t_{\mathsf{P}_2,\text{resp}} &= [8,11]\text{ms} \end{aligned}$$

Obviously, the response times are less than the corresponding periods, and the input interface is met. The output event model can be easily derived. We obtain two periodic event streams with jitter, as described in Section 3.1:

$$\begin{aligned} T_{\mathsf{P}_1,\text{out}} &= T_{\mathsf{P}_1,\text{in}} = 40\text{ms} \\ J_{\mathsf{P}_1,\text{out}} &= t^+_{\mathsf{P}_1,\text{resp}} - t^-_{\mathsf{P}_1,\text{resp}} = 39\text{ms} - 23\text{ms} = 16\text{ms} \\ T_{\mathsf{P}_2,\text{out}} &= T_{\mathsf{P}_2,\text{in}} = 20\text{ms} \\ J_{\mathsf{P}_2,\text{out}} &= t^+_{\mathsf{P}_2,\text{resp}} - t^-_{\mathsf{P}_2,\text{resp}} = 11\text{ms} - 8\text{ms} = 3\text{ms} \end{aligned}$$

Now, we have the output interface of the analysis domain of CPU_1. The next task is to meet the input interface of CPU_2, where minimum inter-arrival times $t_{\mathsf{P}_3,\text{int,in}}$ and $t_{\mathsf{P}_4,\text{int,in}}$ (known from sporadic process activation) are required. Again, we use the EMIF supplied by [14]. In contrast to the timed buffer for the input of P_2, we find out that the interface of CPU_2 can be met without the need of additional interface components. The event stream can be directly input. However, to be able to do the analysis for CPU_2, we have to transform the representation of the event stream. We have periodic event streams with jitter. We find the appropriate EMIF in Table 1 and obtain:

$$\begin{aligned} t_{\mathsf{P}_3,\text{int,in}} &= T_{\mathsf{P}_1,\text{out}} - J_{\mathsf{P}_1,\text{out}} = 24\text{ms} \\ t_{\mathsf{P}_4,\text{int,in}} &= T_{\mathsf{P}_2,\text{out}} - J_{\mathsf{P}_2,\text{out}} = 17\text{ms} \end{aligned}$$

In contrast to the re-synchronization of P_2's input events, no additional components are needed to transform the event stream, as depicted in Figure 4. We just add some math to the models. The response times calculated by the analysis in [2] are:

$$\begin{aligned} t_{\mathsf{P}_3,\text{resp}} &= [13,20]\text{ms} \\ t_{\mathsf{P}_4,\text{resp}} &= [3,15]\text{ms} \end{aligned}$$

Again, we can compute the output event interface. We again propagate the input model to the output and account for the above mentioned "internal" jitter. However, we have to *subtract* the internal jitter, since we are looking for *minimum* inter-arrival times:

$$\begin{aligned} t_{\mathsf{P}_3,\text{int,out}} &= t_{\mathsf{P}_3,\text{int,in}} - (t^+_{\mathsf{P}_3,\text{resp}} - t^-_{\mathsf{P}_3,\text{resp}}) = 17\text{ms} \\ t_{\mathsf{P}_4,\text{int,out}} &= t^-_{\mathsf{P}_4,\text{resp}} = 3\text{ms} \end{aligned}$$

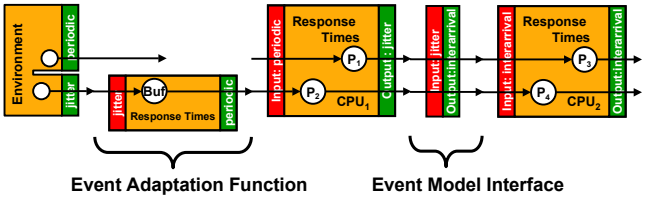


Figure 4: Use of EMIFs and EAFs

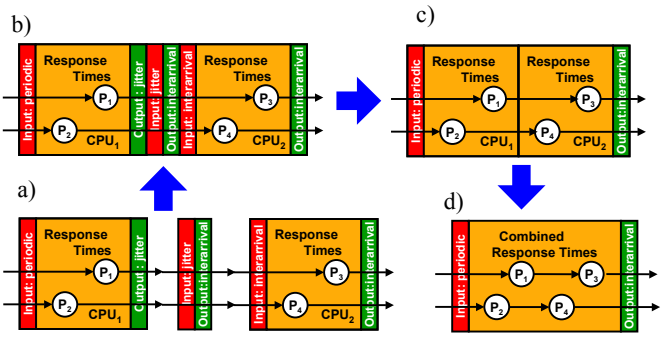


Figure 5: Composition of Analysis Domains

5.2 Composition

Now, after we have successfully added EMIFs and EAFs to the system, all output→input event interfaces match. In other words, no more adaptation is required, and the domains can be directly connected together, as shown in Figures 5(a) and (b). Now, we can eliminate all internal interfaces in our representation (Figure 5(c)) and calculate the accumulated response times along each path:

$$\begin{aligned} t_{\mathsf{P}_1\to\mathsf{P}_3,\text{resp}} &= t_{\mathsf{P}_1,\text{resp}} + t_{\mathsf{P}_3,\text{resp}} \\ t^+_{\mathsf{P}_1\to\mathsf{P}_3,\text{resp}} &= 59\text{ms} \\ t^-_{\mathsf{P}_1\to\mathsf{P}_3,\text{resp}} &= 36\text{ms} \\ t_{\mathsf{P}_2\to\mathsf{P}_4,\text{resp}} &= t_{\mathsf{P}_2,\text{resp}} + t_{\mathsf{P}_4,\text{resp}} \\ t^+_{\mathsf{P}_2\to\mathsf{P}_4,\text{resp}} &= 26\text{ms} \\ t^-_{\mathsf{P}_2\to\mathsf{P}_4,\text{resp}} &= 11\text{ms} \end{aligned}$$

We obtain a system, that is characterized by an input event interface (same as obtained for the original analysis domain CPU_1), an output event interface (obtained for domain CPU_2), and response times for the propagation of events. Following the definitions in 4, we obtain a single analysis domain for our entire sub-system (Figure 5(d)). This single domain can be further hierarchically combined with other domains, and so on. By this, we are now able to apply compositional timing analysis, which is well known from hardware design, to heterogeneous and highly dynamic preemptive software systems.

So far, we have shown, how analysis domains can be coupled and composed into larger analysis domains. We did so by selecting appropriate EMIFs and EAFs to adapt the event models and streams, respectively. We showed that internal EMIFs can eliminated after domain composition, since they do neither add functionality nor architecture to the system. They are just used to couple the analysis approaches. EAFs have to be treated differently when embedded within a newly composed analysis domain. First, they add functionality (buffering) which consumes time. Second, they add architecture (buffers, timers). These influences have to be considered during the composition process.

The pseudo-algorithm for the composition of analysis domains concludes this section. We assume that the scheduling domains (architecture components including RTOS or bus protocols) and the corresponding available analysis techniques have already been identified. Then, we can can define the analysis as an iterative process:

1. select scheduling domain for which all input event models are already determined
2. select analysis approach for scheduling domain (to obtain analysis domain)
3. if necessary, adapt event models (using EMIFs and EAFs)
4. determine process (or communication) response times and output event interfaces from corresponding analysis domain
5. determine output event interfaces
6. repeat from step 1 until all scheduling domains have been analyzed

With wisely selected analysis approaches and event interfaces for the domains, this algorithm will find a solution, if there exists one

with respect to the compositional approach. In practice, one usually needs to back-track several times and modify these selections. Finally, the domains can be merged as shown in Figure 5.

5.3 Feed-Back

So far, only a simple feed-forward example was investigated. Selection of analysis methods was straight forward. Often, several analysis approaches are avaliable for a given problem. As previously mentioned, they substantially differ in the restrictions they impose on the input event interface. And they differ in complexity. Usually, the simpler the analysis, the more limited is the supported input event interface, and, as a result, the higher is the event adaptation overhead in terms of buffers, timers, etc. A reasonable design flow should balance between analysis complexity and input interface limitations. However, in the beginning of the analysis, it is often not clear which analysis is most suitable. Thus, similar to the actual design, also the analysis might consist of several cycles of selecting, rejecting and backtracking of analysis decisions. Even in feed-forward systems, several analysis methods might be applicable for a given problem. However, re-analysis becomes really important when we have more complex systems including *cyclic* dependencies.

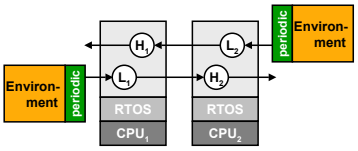


Figure 6: Example System with Cyclic Timing Dependencies

An example for a cyclic dependency of process behavior is given in Figure 6. The output of P_{L_1} (Low priority on component 1) activates P_{H_2} (High priority on component 2) which itself preempts P_{L_2} whose output activates P_{H_1}. And with P_{H_1} preempting P_{L_1}, all processes cyclically depend on each other with respect to timing. Note that this is a rather artificial example. However, similar but less obvious situations are likely to be found in large real-world designs. The challenge of resolving cyclic dependencies will be shown using a very simplified example.

From the specification, we know that the input events of P_{L_1} and P_{L_2} arrive periodically with the periods $T_{P_{L_1},\text{in}}$ and $T_{P_{L_2},\text{in}}$. Furthermore, we assume constant core execution times which are small enough not to violate the input event interface of Liu and Layland's analysis. In the feed-forward example from Sec. 5.1, we started the analysis from system inputs to system outputs by propagating event models. This is not possible here, since there is no dedicated starting point with all parameters given. Hence, we need to start the analysis with making assumptions on the unknown events.

We may start simply by assuming periodic input events for CPU_1. We work from P_{L_1}. The period of P_{H_1} is not yet known. Thus, we just ignore P_{H_1}. As a result, P_{L_1} is not preempted, and its output is periodic, too. Now, we can analyze the timing behavior of CPU_2. We know the two input event models (both periodic), and analyze the behavior. We find out that P_{L_2} can be preempted by P_{H_2}, resulting in a jitter for the output of P_{L_2}, and our initial assumption about periodic input models for P_{H_1} is invalidated. Thus, we reject our initial assumption and try again starting with periodic events with jitter. This procedure has to be continued until the input event interfaces converge. This is not necessarily the case. We carried out two simple experiments. In both experiments, the input periods of the environmental events are $T_{P_{L_1},\text{in}} = 20\text{ms}$ and $T_{P_{L_2},\text{in}} = 30\text{ms}$. We use a very simplistic response time analysis for demonstration. At each activation, we iteratively calculate the worst-case number of process preemptions similar to the approach in [10]. The best-case number of preemptions is set to zero. Experiment one assumes the core execution times of all four processes to be 9ms, while these are 11ms in experiment 2. We start at P_{L_1} and follow the dependencies until we reach a convergence point, i. e. we meet a previously assumed event interface after one iteration. At each step, we calculate new internal jitters. The results of experiment 1 can be seen in Table 3. We see, that the algorithm terminates after three steps, i. e. the input jitters of P_{H_1} and P_{H_2} do not change anymore. Experiment 2 (without table) does not converge. The input jitters of P_{H_1} and P_{H_2} alternatingly increase and can not be bounded. Hence, the input interface assumed by the selected analysis is not met.

Step	$J_{P_{H_2},\text{in}} = J_{P_{L_1},\text{out}}$	$J_{P_{H_1},\text{in}} = J_{P_{L_2},\text{out}}$	comment comment
INIT	0	—	P_{L_1} not preempted
1	0	**9**	P_{L_2} preempted by P_{H_1}
2	**9**	9	P_{L_1} preempted by P_{H_2}
TERM	9	**9**	P_{L_2} preempted by P_{H_1}

Table 3: Experiment 1: $t_{P_i,\text{core}} = 9\text{ms}$

The two experiments show that, while purely feed-forward event propagation through properly described local analysis domains will always lead to a solution, the feedback situation is more complicated (as it is in practice). There are cases where the feedback approach used in the example does not lead to a solution (i. e. a convergence), even though there exists a valid schedule in practice. This can be proved by analyzing the average processor utilization. With 91,67% ($\frac{55}{60}$), the system is in fact schedulable, i. e. there is a point in time where total system workload will be zero and buffers are empty. However, response time analysis is more complicated than only guaranteeing schedulability.

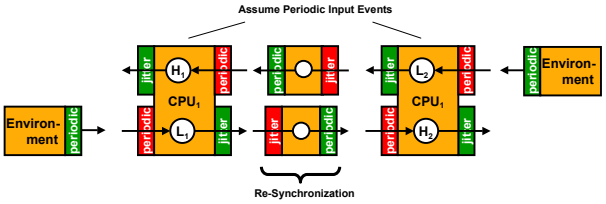


Figure 7: Cyclic Dependencies Resolved by Re-Synchronization

This effect is known from schedulability analysis and is due to the abstraction underlying the event propagation. It is often possible to enforce convergence by inserting buffers for synchronization, as depicted in Figure 7. Here, analysis domains with periodic input event interfaces can be used. Trivially, the iterative propagation of event models terminates after the first cycle. The introduction of buffers to decouple strong interdependencies is what an experienced designer would do in a manual design. As a key advantage of our approach, we can automatically synthesize the required adaptation function for a given analysis domain.

A similar situation can be found when the processes graph itself contains cycles, e. g. feed-back loops in control engineering. The same problems apply. However, cyclic process graphs are much better understood than cyclic non-functional dependencies. Designers are used to insert buffers to store events within the cycle in order to avoid deadlock. Finding a general theory for the feedback case is part of our current research.

6. EXAMPLE

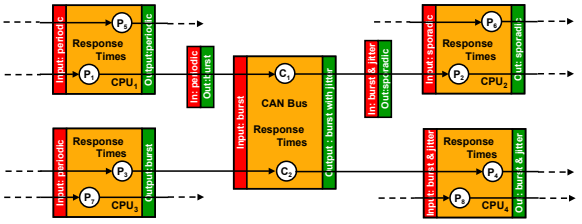


Figure 8: Example of Four Processors and One Bus

Figure 8 shows the analysis domains of a complex heterogeneous platform architecture. It consists of four processors which are connected via a CAN bus [12]. Processor CPU_1 runs a deterministic static scheduler. CPU_2 runs a round robin scheduler. The other processors run static-priority schedulers. The arbitration of packets on the CAN bus is based on static priorities for communicated packets. We have already select appropriate analysis approaches for each of the five analysis domains.

Process P_1, implemented on CPU_1, sends data to process P_2 on CPU_2 over the communication channel C_1. Similarly, processes P_3 and P_4 exchange data via C_2. The other four processes represent the resource sharing influences on the processors. However, they do not use the bus.

From the environment (indicated by the dashed lines) we know that P_1 is activated periodically ($T_{P_1,\text{in}} = 10\text{ms}$). The timing behavior of CPU_1 is completely determinate, e. g. periodic execution of simple digital signal processing functions with a fixed response time ($t_{P_1,\text{resp}} = 4\text{ms}$). Thus, the output is purely periodic, too ($T_{P_1,\text{out}} = 10\text{ms}$). P_3 is also activated periodically ($T_{P_3,\text{in}} = 10\text{ms}$). However, is has a low priority on CPU_3 and is heavily preempted by P_7 resulting in a bursty output behavior ($T_{P_3,\text{out}} = 100\text{ms}, t_{P_3,\text{int,out}} = 2\text{ms}, b_{P_3,\text{in}} = 10$). On the bus, the channel C_2 has high priority, while C_1 has low priority. We are interested in the timing behavior of the path $P_1 \rightarrow C_1 \rightarrow P_2$.

In general, two corner-case situations on the bus can be distinguished:

- P_3 outputs a burst of packets to the bus. Since the corresponding channel C_2 has a higher priority, the (lower-priority) packets from P_1 are blocked until the end of the burst. The communication delay of channel C_1, and thus, the overall latency of the path under consideration is high.
- P_3 does not output any packets for some time between two bursts (because it is preempted by P_7). Packets coming from P_1 are not blocked, communication delay on C_1 is low, and the path latency will be low, too.

First, we have to analyze the bus before we can analyze the behavior of P_2. We apply our event model interfacing technique to adapt the event models between P_1 and C_1. From Table 1, we obtain the burst representation of P_1's output: $T_{C_1,\text{in}} = 10\text{ms}, t_{C_1,\text{int,in}} = 10\text{ms}, b_{C_1,\text{in}} = 1$. The transmission of each packet over the bus constantly requires 5ms. For the analysis, we can directly use Lehoczky's [8] approach. The fact that single bus packets can not be preempted regardless of the priority is considered assuming semaphore locking. As a result, we obtain:

$$\begin{aligned} t_{C_1,\text{resp}} &= [5\text{ms}, 30\text{ms}] \\ t_{C_2,\text{resp}} &= [5\text{ms}, 28\text{ms}] \end{aligned}$$

We see that, although the C_1-packets are to be sent over the bus periodically, the packet response time heavily varies. The best- and worst-case packet response times correspond to the above distinguished situations. From the analysis, we also obtain C_1's output behavior. There are $b_{C_1,\text{out}} = 10$ packets within an outer period of $T_{C_1,\text{out}} = 100\text{ms}$. This results from the influence of C_2-bursts. The minimum inter-arrival time of communicated packets equals the packet transmission delay ($t_{C_1,\text{int}} = 5\text{ms}$). In other words, the bursty behavior on C_2 turns the periodic send-requests on C_1 into burst communication with similar characteristics.

Now, we can analyze the behavior of P_2. The event model interface (EMIF) between C_1 and P_2 is taken from Table 1, and the analysis of the round robin scheduled processes on CPU_2 results in a response time interval for P_2:

$$t_{P_2,\text{resp}} = [7\text{ms}, 15\text{ms}]$$

It remains to determine the overall event propagation delay along the path $P_1 \rightarrow C_1 \rightarrow P_2$:

$$\begin{aligned} t_{P_1 \rightarrow C_1 \rightarrow P_2,\text{resp}} &= t_{P_1,\text{resp}} + t_{C_1,\text{resp}} + t_{P_2,\text{resp}} \\ &= 4\text{ms} + [5\text{ms}, 30\text{ms}] + [7\text{ms}, 15\text{ms}] \\ &= [16\text{ms}, 49\text{ms}] \end{aligned}$$

7. CONCLUSION

We have presented a novel compositional approach for scheduling analysis in platform design. We identified analysis domains, i. e. architecture components, scheduling strategies, and appropriate analysis approaches. Using the concept of abstract event adaptation and propagation, we were able to couple initially incompatible scheduling analysis approaches. After coupling, we hierarchically combined the analysis domains. As a result, we were able to formally analyze the timing behavior of complex platforms, combining several preemptive and non preemptive scheduling strategies in one system. The variety of avaliable event model interfaces and adaptation functions shows the generality of our approach.

Our contribution provides a new quality of timing verification in platform design. Conservative bounds on system response times can be provided for highly complex, heterogeneous platforms. We are currently implementing the analysis procedure presented in this paper, including a repository for efficiently storing, identifying, and executing the existing analysis approaches.

Our current research in this area includes but is not limited to finding heuristics to resolve cyclic analysis domain dependencies, and extending the set of event adaptation functions.

8. REFERENCES

[1] Cadence. *Cierto VCC Environment*. http://www.cadence.com/products/vcc.html.

[2] Beatriz Asensio Calvo. Real-time analysis of time-driven scheduling – a quantitative comparison of Round Robin and TDMA. Technical report, Technical University of Braunschweig, September 2001.

[3] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *Proceedings of International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 16–30, 1998.

[4] K. Gresser. An event model for deadline verification of hard real-time systems. In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 118–123, Oulu, Finland, 1993.

[5] A. Hergenhan and W. Rosenstiel. Static timing analysis of embedded software on advanced processor architectures. In *Proceedings of Design, Automation and Test in Europe (DATE '00)*, pages 552–559, Paris, March 2000.

[6] H. Kopetz and G. Gruensteidl. TTP - a time-triggered protocol for fault-tolerant computing. In *Proceedings 23rd International Symposium on Fault-Tolerant Computing*, pages 524–532, 1993.

[7] David C. Ku and Giovanni de Micheli. *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[8] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings Real-Time Systems Symposiom*, pages 201–209, 1990.

[9] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[10] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] Mentor Graphics. *Seamless Co-Verification Environment*. http://www.mentorg.com/seamless/.

[12] Philips Semiconductors. *Controller Area Network CAN*. http://www.semiconductors.philips.com/can/.

[13] P. Pop, P. Eles, and Z. Peng. Bus access optimization for distributed embedded systems based on schedulability analysis. In *Proc. Design, Automation and Test in Europe (DATE 2000)*, Paris, France, 2000.

[14] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proc. of Design, Automation and Test in Europe Conference (DATE'02)*, Paris, France, March 2002.

[15] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[16] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1(1):27–60, 1989.

[17] F. Wolf and R. Ernst. Execution Cost Interval Refinement in Static Software Analysis. *Journal of Systems Architecture*.