Scheduling Analysis Integration for Heterogeneous Multiprocessor SoC

Kai Richter, Razvan Racu, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
D-38106 Braunschweig / Germany
{**richter|racu|ernst**}**@ida.ing.tu-bs.de**

Abstract

Today, only very few techniques out of the host of work on formal performance and timing analysis have been adopted in MpSoC (multiprocessor system-on-chip) design. One of the key reasons is a mismatch between the scheduling models assumed in most formal approaches and the heterogenous world of MpSoC scheduling techniques and communication patterns. This heterogeneity results from IP reuse and a plug-and-play *design style, required to effectively reach the necessary design productivity. A second problem is the model complexity. While complex, specialized models can find their way into industry niches, their broad acceptance is extremely doubtful. In this paper, we review the existing scheduling analysis techniques with respect to these key requirements and derive a good compromise between model simplicity on the one hand, and applicability to MpSoC design on the other hand. The approach represents system-level scheduling analysis as a flow-analysis problem for event streams that can be configured to reuse the existing local scheduling analysis techniques. We define transformations between few key event stream models to meet the interfacing requirements of the compositional design style. An example demonstrates the application of the approach, as well as the worthiness of the results.*

1. INTRODUCTION

With increasing embedded system design complexity there is a trend towards heterogeneous architectures. Todays high-end multiprocessor systems-on-chip (MpSoCs) integrate multiple programable processor cores, specialized memories, and other intellectual property (IP) components on a single chip using complex networks on-chip (NoC). Several operating systems and bus protocols can be found on such MpSoCs.

Heterogeneous MpSoCs have become the architecture of choice in major industries such as network processing, consumer electronics, and automotive systems. Since no homogeneous design strategy is sufficiently optimal for all aspects of MpSoC, their heterogeneity inevitably increases with IP integration and component specialization which designers use to optimize performance at low power consumption and competitive cost.

Figure 1 shows an example MpSoC, the Viper [9] processor for multimedia applications. Based on the Philips Nexperia platform, it includes two cores, weakly programmable coprocessors, and fixed-function coprocessors, as well as various memories and caches omitted in the figure. A complex network of bridged high-speed and peripheral buses connect these components.

Many key components are either reused or supplied externally, such as the MIPS and TriMedia processor cores. Tomorrows MpSoCs will be even more complex, and using such IP library elements in a *plug-and-play* design style is the only way to reach the necessary design productivity.

Hence, systems integration is becoming the major challenge in MpSoC design. The complex hardware and software component interactions –including heterogeneous scheduling environments– pose a serious threat to all kinds of performance pitfalls, including transient overloads, memory overflow, data loss, and missed deadlines. The *International Technology Roadmap for Semiconductors* (ITRS, [25]) names system-level performance verification as one of the top three codesign issues.

It might surprize that –up to now– only very few of the countless formal analysis approaches from the real-time community have found their way into the SoC (system-on-chip) design community by means of tools. Regardless of the known limitations of simulation such as incomplete corner-case coverage and pattern generation, timed simulation using e. g. Mentor Graphics Seamless-CVE [20], Axys MaxSim [4], or Cadence VCC [6] is still the preferred means of performance verification in MpSoC design. But why is the acceptance of formal analysis still very limited?

One of the key reasons is a mismatch between the scheduling models assumed in most formal analysis approaches and the heterogenous world of MpSoC scheduling techniques and communication patterns that are a result of a) different application character-

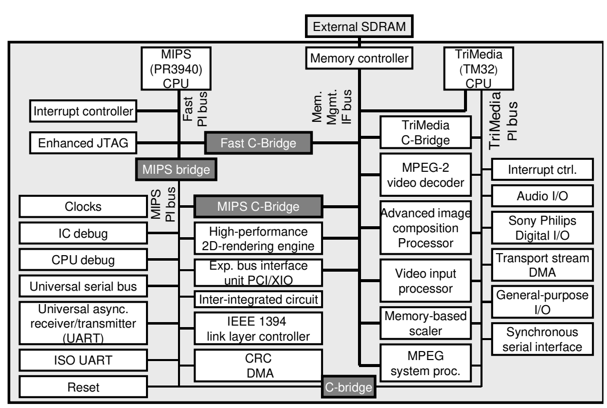


Figure 1: The VIPER Processor

istics; b) system optimization and integration just as shown in the example of Fig. 1 which is still at the beginning of the MpSoC development towards even more complex architectures.

Therefore, a new configurable analysis process is needed that can easily be adapted to such heterogeneous architectures. We can identify different approaches: the holistic approach that searches for techniques spanning several scheduling domains; and hierarchical approaches that integrate local analysis with a global flow based analysis, either using new models or based on existing models and analysis techniques.

In the following section, we will more deeply review the existing analysis approaches from the literature on real-time analysis and identify key requirements for their application to MpSoC design. In Section 3, we define a six class event model that serves as a sufficiently optimal compromise between model simplicity and applicability to MpSoC design. Section 4 presents all necessary event model transformations to be fully applicable to the compositional MpSoC integration style. Specialties of event stream dependency cycles are investigated in Section 5. A set of expressive experiments is carried out in section 6. We interpret the experimental results, before we draw our conclusions.

2. KNOWN APPROACHES

2.1 The One-Model Approaches

Some well established classical analysis techniques such as Rate-Monotonic Scheduling [19] (RMS) consider a system of independent tasks, and use simple task activation models, such as periodic activation. More complex activation can be considered using periodic tasks with the so called *release jitter* [2], *sporadic tasks* [27], and *sporadically periodic* tasks [2], also referred to as *sporadic bursts* [32]. Based on these models, efficient schedulability tests [19, 18] and *response time* algorithms have been proposed [14, 17, 2, 32].

$$R_i = C_i + \sum_{j \in HP(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq T_i \qquad (1)$$

Equation 1 shows the popular RMS response time (R_i) approach [14], where C_i and T_i are the *core execution time* and the period of task i, and $HP(i)$ is the set of all higher priority task s. The equation can be iteratively solved using $R = 0$ as a starting point [2]. The periodic nature of task activation is accounted for in the term

$$\left\lceil \frac{R_i}{T_j} \right\rceil$$

determining the number of (periodic) preemptions due to higher level task j during the response time of task i.

A similar set of analysis techniques is available for EDF (earliest deadline first) scheduling [19, 13]. An extensive overview on EDF scheduling and analysis can be found in [28]. Time-driven techniques such as TDMA (time division multiple access) and RR (Round Robin) have also been investigated [15, 16].

Such elegantly simple approaches already found their way to practice some time ago, and there exist today an increasing number of commercial analysis tools such as TriPacific's RapidRMA [34], TimeWiz [30] from TimeSys, and Livedevices' Real-Time Architect [10], all targeting at system load and process response time analysis. Communication network IP such as TTP [35] and Sonics [26], supported by appropriate optimization tools based on formal analysis, is becoming available. This shows that the systems community is in fact seeking the assistance of formal real-time analysis. Unfortunately, homogeneous scheduling strategies are assumed by these relatively simple approaches, so they do not scale to large, heterogeneous systems.

There exist few approaches considering larger systems such as [21, 22, 33, 31] analyzing and optimizing static priority task scheduling combined with a TDMA bus protocol. The authors of [28] extend this to EDF task scheduling.

These holistic analysis techniques are certainly very effective for their respective applications. There are standard architectures such as in automotive communication where it makes sense to develop such specialized techniques. However, holistic approaches require a model complexity that grows with the size of the systems and with the number of different scheduling techniques. New combinations require new holistic models which counters the need for flexible integration and rapid design space exploration. This could explain, why such holistic approaches are largely ignored by the SoC community even though there are many proposals for multiprocessor analysis in real-time computing.

Other, very sophisticated approaches like *timed automatas* [1] or *modal processes* [8] consider the system as the cross product of all subsystem states (or modes) that result from scheduling. These models are even more complex than the mentioned holistic ones, and analysis algorithms generally suffer from exponential complexity. We are not aware of an application of such approaches to heterogeneous SoC. Therefore, they will not be further considered here.

Analysis techniques and their configuration to an individual heterogeneous MpSoC should be simple, efficient, and follow the integration of the hardware and software components just like it is done with the combination of models for simulation. It should follow the compositional design style which has been adopted in MpSoC design to reach the tremendous design productivity requirements. Such a feature would support the introduction of formal analysis to MpSoC design.

Compositional design requires interfaces that describe communication between components and subsystems. When formulating such interfaces, it appears reasonable to resort to the body of knowledge in intuitive event models developed in the real-time systems community. We will first review scheduling techniques and their respective event models before we propose such an interface.

2.2 Model Commonalities

The accepted models have a key commonality. They use intuitive event models! The assumption of a periodic event with jitter even lets an experienced engineer create worst-case task activation scenarios. And this is exactly what the formal techniques do. As seen in Equation 1, they calculate a maximum number of task activations for a given amount of time, to be used in accumulative worst-case response time equations.

Quite similar, but less often fully understood, are the minimum numbers required for best-case analysis to resolve scheduling anomalies in distributed systems [11].

Table 1: The n_{act}-functions of the four most popular event models

model	params	$n^{+}_{\mathrm{act}}(\Delta t)$	$n^{-}_{\mathrm{act}}(\Delta t)$
periodic	$< T >$	$\lceil \frac{\Delta t}{T} \rceil$	$\lfloor \frac{\Delta t}{T} \rfloor$
jitter	$< T, J >$	$\lceil \frac{\Delta t + J}{T} \rceil$	$\max\left(0, \lfloor \frac{\Delta t - J}{T} \rfloor\right)$
sporadic	$< t >$	$\lceil \frac{\Delta t}{t} \rceil$	0
burst	$< T, t, b >$	$\lfloor \frac{\Delta t}{T} \rfloor b + \min\left(b, \left\lceil \frac{\Delta t - \lfloor \frac{\Delta t}{T} \rfloor T}{t} \right\rceil\right)$	0

Table 1 gives an overview about these n^{+}_{act}- and n^{-}_{act}-functions (maximum and minimum number of activations) for the four most popular event models in literature: periodic [19, 18, 3] (used in Equation 1), periodic with jitter [2], sporadic [27, 2], and burst [33] (also known as sporadically periodic [2]).

The known techniques, especially the ones from Lehoczky [17] and Tindell [32], which allow arbitrary deadlines and are thus less constrained, could potentially deal with *any* event model, as long as the n_{act}-functions are known.

2.3 Subsystem Composition

So far for the commonalities among the majorly accepted models. We will now briefly review recent contributions which – in contrast to *single component* or *holistic* approaches– heavily exploit the existence of such n_{act}-functions for a *compositional* analysis approach.

Gresser [12] and Thiele [29] established a different view on scheduling analysis. The individual components or subsystems are seen as entities which interact, or communicate, via event streams. In their compositional approach, an output event stream of one component turns into an input event stream of a connected component. Schedulability analysis, then, becomes a flow-analysis problem for event streams that, in principle, can be solved iteratively using event stream propagation.

Both approaches use a highly generalized event stream representation to tame the complexity of the event streams. Gresser uses a superpositional *event vector system*, which is then propagated using complex event dependency matrices. Thiele et. al. use a more intuitive model. They use *numerical* upper and lower bound event *arrival curves* for event streams, and similar *service curves* for execution modeling. Linear approximation [7] is used to feed the curves into an existing recurring task approach [5]. The approach has already been adopted to network processor design, where numerical (or statistical) stream modeling is commonly used. But it suffers from its missing abstraction level, since its is not straight-forward to re-extract key event model properties such as a period –required in most real-time scheduling techniques– from these numerical curves. Finally, the approach is currently limited to worst case curves (due to the used local analysis techniques) such that scheduling anomalies as described above are not covered.

But both compositional approaches are a good starting point for the following considerations. They use some event stream representation to allow component-wise local analysis. The local analysis results are, then, propagated through the system to reach a global analysis result.

2.4 Models and System Composition

After reviewing a representative subset of formal real-time analysis, we can rephrase our analysis requirements. The formal compositional analysis should:

- support heterogeneous systems with different component and subsystem scheduling strategies, since current MpSoC look like this.
- capture communication via event models that are suitable to allow the local application of standard analysis techniques because these are industrially accepted.
- do this in a form that encompasses the industrial *plug-and-play* systems integration style.

Our proposal closely follows the pros in the above overview. We will use the simple event models (periodic, jitter, sporadic, and burst) and apply the compositional idea of Thiele and Gresser to them.

This procedure reveals two previously unrecognized challenges:

- As a prerequisite for composition based on event streams, **component output streams** need to be determined using only the simple (comprehensible) event models.
- Additionally, each individual SoC component can **constrain the acceptable input stream** to a particular model, e. g. because of a fixed periodic scheduling (as found in DSP applications), or due to IP (intellectual property) protection. This requirement arises from the industrial *plug&play* integration style.

These challenges are not specific to our proposed approach. Rather, they are key to generally making the compositional analysis comprehensible *and* applicable to current industrial designs.

3. OUTPUT EVENT MODELS

In the compositional approach, events are seen as *traveling* through a network of components, thereby triggering task and communication execution.

Each input event experiences a delay when traveling through a component. With a constant delay, the relative timing of events in the stream remains constant, only the absolute time offset changes from input to output. Thus, the event model does not change, either. However, resource sharing –irrespective whether processor, bus, or network– induces non-constant delays. This is reflected by a *response time interval*, constrained by upper and lower response time bounds. This introduces uncertainty to the output timing, we can not predict the exact event arrival time, anymore. In other words, a non-constant delay adds jitter characteristics to the event stream.

An example system is shown in Figure 2 interating two functionally independent subsystems via a shared bus or network (NoC). The purely periodic data stream coming from the IP_1 component enters the network on channel C_3. Communication scheduling analysis can use the simple model of periodic events to capture this communication. Due to interference with the other channels, the data on channel C_3 experiences non constant network delays. Hence, the data arrives at the DSP generally periodic but with a jitter, determined by the difference between the maximum and minimum network delay (or network response time). We can directly use the known jitter

Table 2: The n_{act}-functions of the new models

model	params	constraints	both $n^{+}_{\mathrm{act}}(\Delta t)$	periodic $n^{-}_{\mathrm{act}}(\Delta t)$	sporadic
simple	$< T >$	$T > 0$	$\left\lceil \frac{\Delta t}{T} \right\rceil$	$\left\lfloor \frac{\Delta t}{T} \right\rfloor$	0
w/ jitter	$< T, J >$	$T > J \geq 0$	$\left\lceil \frac{\Delta t + J}{T} \right\rceil$	$\max\left(0, \left\lfloor \frac{\Delta t - J}{T} \right\rfloor\right)$	0
w/ burst	$< T, J, d >$	$J \geq T > 0, \quad d \geq 0$	$\min\left(\left\lceil \frac{\Delta t + J}{T} \right\rceil, \left\lceil \frac{\Delta t}{d} \right\rceil\right)$	$\max\left(0, \left\lfloor \frac{\Delta t - J}{T} \right\rfloor\right)$	0

event model [2] to analyze the DSP scheduling. In effect, the network just transformed the stream from one *well known* model into another, and existing analysis techniques can be safely applied to each component.

Unfortunately, this is not always the case, as the question marks at the outputs of the remaining network channels, C_1 and C_2, in Figure 2 show. The already *jittered* output from the CPU will experience additional distortion on the network, possibly resulting in a jitter that exceeds the period. But in many analysis approaches which support periodic events with jitter [2], the jitter must not exceed the given period. However, the above example shows that every component potentially increases the jitter, so at some point, we would need to transition to Tindell's model of sporadic bursts, because we do not know of a model of periodic events with burst.

Surprisingly, this transition is relatively complex, since Tindell's model is not intended to provide large jitter support. Based on sporadic events, it has to be treated independent of the known periodic models. Hence, we further lose the generally periodic nature of the stream. This is, as we will see later in Section 4, one of the key event stream properties with respect to analysis accuracy and adaptability.

As another example, look at the sporadic event stream from the sensor Sens. The network adds jitter characteristics, but there is no model of "sporadic events with jitter". Again, we would need to conservatively transition into sporadic with burst.

This lack of appropriate output event models shows that such effects have received only little attention in literature, so far. In [24], we developed complex propagation functions to stay within the four models. However, we finally decided to define a slight, intuitive extension into a comprehensive set of six models as a better match to the simulation pattern sequences that SoC designers are used to.

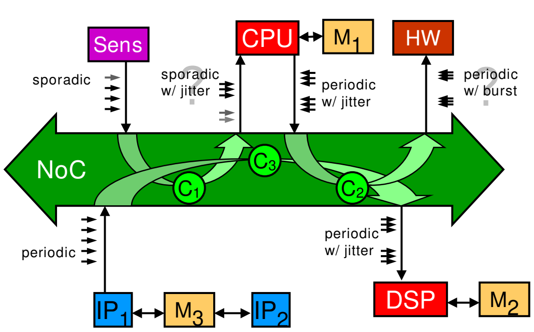


Figure 2: Propagation Jitter due to Network Arbitration

3.1 A Simple Self-Contained Event Model Set

We define two classes, *periodic* and *sporadic*, and we have three models in each class: *simple*, *with jitter*, and *with burst*.

That means that we keep on with periodic and periodic with jitter, but define a *burst* as a *jitter, that exceeds the period*. In order to limit the maximum transient frequency, we additionally define a minimum distance d between any two events, similar to Tindell's *inner period* [32]. Table 2 contains the n_{act}-functions of the models.

The *sporadic* class generalizes the class of *periodic* models, in that only the n^{-}_{act} functions are set to zero. That means, every sporadic model can be assumed to be the corresponding periodic model in the worst case, which goes along with the early applications of sporadic events in real-time analysis [27].

There are three advantages of these six models over the four previously identified *well known* ones. First, the transitions from *simple* to *jitter* to *burst* is even more intuitive than it was possible with Tindell's burst model. It just accounts for large jitters. Secondly, the periodic and sporadic models use the same underlying models, which further reduces the complexity.

However, the major advantage of this new set of models is a conceptional one: The new set of models is self-contained with respect to the previously mentioned propagation step. In other words, no propagation brings us out of these six models, which was the first major challenge mentioned in Section 2.4. Jitters are allowed to exceed the period far over, and our new definition of *burst* is just the intuitive characterization of the effects of large jitters. This sufficiency of a bounded set of intuitively structured models is likely to attract MpSoC designers attention to the overall ideas of formal real-time analysis.

With respect to the previous example (Fig. 2), we can now represent all event streams using only the six new models: channel C_1 outputs *sporadic events with jitter* to the CPU, while channel C_2 turns the *periodic with jitter* at the CPU output into a *periodic with burst* at the input of HW.

This six class model appears as an efficient compromise between model simplicity and intuition on the one hand, and completeness on the other hand. As we have explained in Sec. 2.2, it can be directly – without any approximation– applied in connection with established local analysis techniques which we consider a major advantage.

It should, however, be mentioned that the six models still do not fully cover arbitrary event stream properties. This is an almost unavoidable limitation, since the extensions to more general models exhibit conceptional disadvantages, as mentioned earlier in Section 2.3. It will, therefore, not be further considered, here.

4. EVENT MODEL TRANSFORMATIONS

In the previous section, we solved the Challenge 1 where we considered the specialties of component output streams. Now, we tackle the second challenge identified in Section 2.4, that is concerned with component input streams. We want to underline that this is not a specific problem of the newly defined event models. It also exists for any other set of models [23]. However, the new set of models allows much more elegant solutions.

As already mentioned in Section 2.4, there are –in the SoC design world– situations, where the input stream of some component is constrained to meet a particular models, as shown in Figure 3(a) . If the input stream is represented using another model, the analysis can not be directly applied. However, many of such seeming incompatibilities can be solved by event model transformations.

We distinguish two types of transformations: those transformations that only transform the formal stream representation, we call them Event Model Interfaces (EMIF); and those which require to also adapt the timing of the stream to find an EMIF, we call them Event Adaptation Functions (EAFs).

4.1 Event Model Interface–EMIF

We start introducing EMIFs using the already known example, but this time, we focus on the input of the HW component. From Section 3, we know that the input stream is periodic with burst. Let us now assume that the HW component is an IP component, i. e. we do not know the internal details nor did the IP supplier give us an analysis. We just know –possibly from the components specifications– that there exists a maximum allowed input frequency, a situation which is not uncommon for HW coprocessors.

The information of a maximum frequency directly translates into a minimum distance of input events, which –in turn– equals a *simple sporadic* event model, as defined in Section 3.1. We see, that the given model is periodic with burst, so the models seem incompatible. But in this example, we can derive the required simple sporadic parameter of the *target model* from the parameters of the given periodic with burst *source model*. Moreover, the minimum distance d of the burst model already is the sought-after parameter t (minimum interarrival time) for the sporadic model ($t = d$).

We can formally prove this through the n_{act}-functions from Table 2. We have to guarantee that the following two equations hold true:

$$n^{+}_{\text{act,source}}(\Delta t) \overset{!}{\leq} n^{+}_{\text{act,target}}(\Delta t) \tag{2}$$

$$n^{-}_{\text{act,source}}(\Delta t) \overset{!}{\geq} n^{-}_{\text{act,target}}(\Delta t) \tag{3}$$

These *event model compatibility tests* check whether the source stream behavior is covered by the target stream model, if both –the actual stream, and the required model– are fully specified.

We have shown that finding the right target model parameters can be complex when allowing arbitrary models [23]. But within our self-contained set of models, most transformations are relative simple and straight-forward. This is because we –in contrast to the work in [23]– are consistently extending two simple *base models* (periodic and sporadic), rather than allowing arbitrary model combinations.

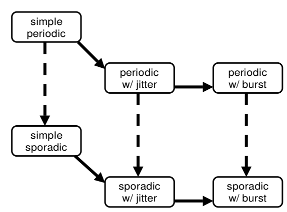


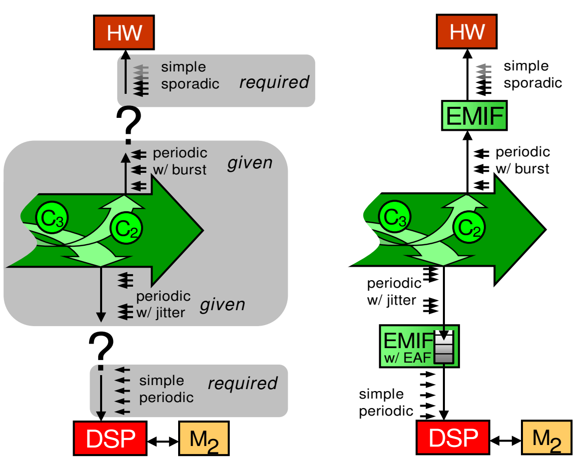
Figure 4: Existing Lossy and Lossless EMIFs in the General Case

Basically, an EMIF is always possible, when the target model is more general than the source model. Two classes of EMIFs can be easily distinguished:

Lossless Transformation. Each base model can be directly transformed into the corresponding models with jitter or burst. The parameters are the same, and any new parameter –a parameter that was not defined in the source model– can be safely set to zero. This is indicated by the solid arrows in Figure 4. In these EMIFs, we can show equality in both equations 2 and 3, and we consider the transformations as *lossless*, i. e. the target model is able to capture the full details of the source stream.

Such transformations can be *losslessly* reversible (not shown in the figure), but only if the source stream represents a *special case* of the target model, e. g. jitter models with the jitter J being zero.

Lossy Transformation. Each periodic model can be directly transformed into the corresponding sporadic event model (indicated by dashed arrows). Such transformations are *lossy* [23], i. e. the target model is less expressive than the source model. Due to the *lossy* nature of these transformations, they are not invertible.



(a) Input Stream Incompatibilities

(b) Event Model Interfaces and Adaptation Functions

Figure 3: Solving Input Event Stream Incompatibilities using EMIFs and EAFs

There are few additional lossy EMIFs such as the transformation from *sporadic with jitter* into *simple sporadic*. The new sporadic period can be calculated as the difference between the source period and the source jitter:

$$T_{\text{target}} = T_{\text{source}} - J_{\text{source}}.$$

And the example at the beginning of this section already demonstrated the transformation from a bursty model into the simple sporadic model.

Clearly, we can combine subsequent transformation steps, e. g. from *simple periodic* to *simple sporadic* to *sporadic with jitter*. However, there are still certain missing EMIFs.

4.2 Event Adaptation Function – EAF

Obviously, we can never transform a sporadic model into a periodic model. The irregular nature [2] of sporadic streams ($n^{-}_{\text{act}} = 0$) makes the compatibility equation 3 fail –except the useless case of a zero period. Similarly, we can –in the general case– not reduce a *burst* into a *jitter* into a *simple* model. But we can *force* a more general event stream into a more constrained model by reducing the uncertainty related to the jitter.

We again start with a simple, intuitive example. Recall the DSP subsystem in Figure 2. The periodic output of IP_1 was distorted by the network and arrives at the DSP input with a jitter. But the DSP requires exactly periodic input to run efficiently as planned. We can simply insert a buffer between the network and the DSP that eliminates the network-induced jitter. As a result, we have reconstructed a purely periodic event stream the DSP can deal with. Similarly, we can *resynchronize* periodic event streams with burst.

In [23], we have shown, how tight buffer sizes can be derived directly from the model parameters[1]. This helps designers to optimize buffer memory which is a limited resource on MpSoCs. Analogously, we can easily derive the upper and lower bound buffering delays which add to the overall event stream latency [23].

To conclude the overview about EMIFs and EAFs, Figure 3(b) shows the integration of EMIFs and EAFs as design elements into the system model. This gives designers an adequate understanding of the effects of system integration under real-time constraints. They do not actually need to understand the transformations in detail, the transformations rather offload the complexity from the systems designers and lets them focus on their top priority problems, i. e. putting the components together.

5. CYCLIC EVENT STREAM DEPENDENCIES

While the above buffering example was straight-forward and intuitive, the insertion of EAFs can be of central concern in systems with cyclic event stream dependencies.

Cyclic data dependencies through feed-back are heavily used for implementing complex filter functions in control or signal processing applications. Designers of such systems intuitively insert resynchronization buffers to break up such cycles.

More complex are situations, where cyclic dependencies are not reflected in the system function. Figure 5 shows an example of a

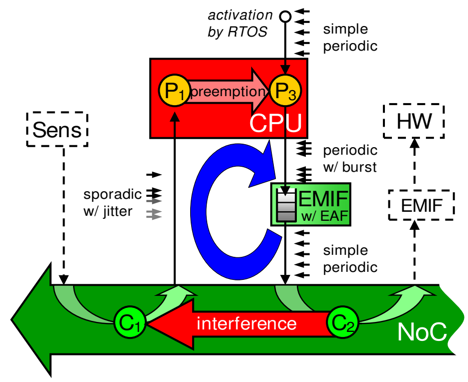


Figure 5: Non-Functional Dependency Cycle

non-functional event stream dependency in the system of Figure 2 which was only introduced by communication sharing between the channels C_1 and C_2.

Upon receipt of new sensor data, the CPU activates process P_1 which preempts P_3 and thus distorts the (initially periodic) execution timing of P_3. P_3's output, in turn, enters the network on channel C_2, where it now interferes with the arriving sensor data on C_1. The interference of the two functionally independent channels, C_1 and C_2, closes the dependency cycle.

Such event flow cycles are not an artificial result of global analysis but exist in practice. They represent a complex performance hazard in MpSoC design, since the dependencies are subtle and difficult to detect with current design practices.

With the approach of Section 3, designers can now analyze such cycles by iterative propagation of event streams until the event stream parameters converge (non-critical cycle) or a deadline or other timing constraint is violated (critical cycle). This iteration process terminates, because the event timing uncertainty –that is the best-case to worst-case response time jitter interval– grows monotonically with every iteration (see Sec. 3).

For cases in which no convergence occurs automatically, we can use EAFs to break up the dependency cycle and enforce convergence by reducing the timing uncertainty, illustrated by the additional EAF buffer in Figure 5. Hence, the approach allows designers to also optimize buffering in far less obvious situations, compared to simple, functional *feed-back* cycles which the individual function designers could fully oversee.

6. EXPERIMENTS

So far, we defined the models and presented event stream model propagation, interfacing, and adaptation based on these models.

Now, we demonstrate the applicability of the approach by fully analyzing the MpSoC example in Figure 6 using only the six event models and a well known analysis techniques, namely Tindell's approach [32].

The experiments will heavily exploit the new ideas, but will at the same time assume a design engineer from the SoC community performing the experiments. In other words, we will hide all the complexity in the local component analysis –which is already accepted by this community–, to demonstrate the tremendous simplification of our proposed approach.

[1] Note that in [23], we used a periodic version of Tindell's specific burst model; however, the key ideas remain.

6.1 Set-Up

The Actuators

There are three actuators: The sensor sporadically sends data blocks of $8kb$ size to P_1, with a maximum sending frequency of $1,7kHz$, which corresponds to a *sporadic event model* with a minimum sporadic period of $\frac{1}{1,7kHz} = 588\mu s$. Process P_3 is periodically activated by the RTOS (real-time operating system) on the CPU with a period of $\frac{1}{20kHz} = 50\mu s$. The high-performance DSP application on IP_1 has a sending frequency of $140kHz$, corresponding to a period of $7,143\mu s$.

The Network

Instead of sending the complete data block, the data packets are fragmented to avoid too long blocking times. Each $8kB$ data block from the sensor is split into 32 packets of $262byte$ each, $256bytes$ plus $6bytes$ protocol overhead–address, length, and CRC. The $3kB$ blocks from P_3 are split into 24 packets of $(128+6) = 134bytes$. This channel C_2 has a higher priority than channel C_1. The highest-priority channel C_3 does not split the DSP data packets, but only adds the $6byte$ protocol information.

The overall average network load L_{net} is:

$$\begin{aligned} L_{\text{net}} &= L_{\text{sensor}} + L_{\text{CPU}} + L_{\text{DSP}} \\ &= 14,2528Mbyte/s + 64,32Mbyte/s + 144,2Mbyte/s \\ &= 222,77Mbyte/s \end{aligned}$$

Execution and Communication Times

For simplicity, the *core execution times* of the two processes on the CPU are assumed constant: $250\mu s$ for P_1 and $10\mu s$ for P_3. This is not a limitation of the approach but rather a clarification of the following tables.

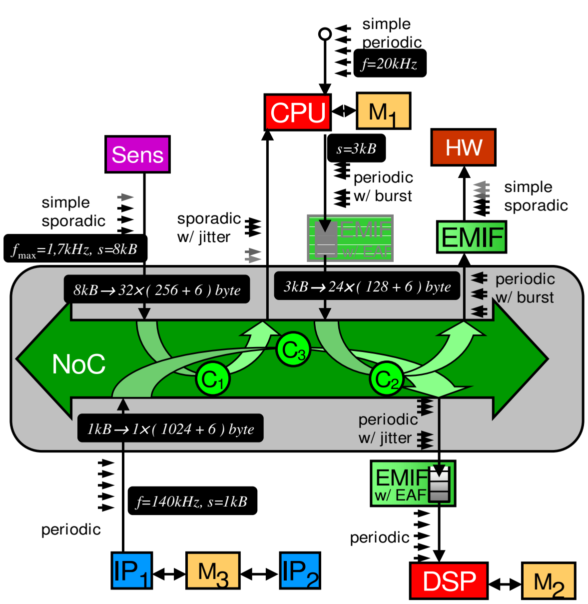


Figure 6: Example System with EMIFs and EAFs

The communication times of the network packets depend on the network speed. Since we perform several experiments with different network speeds, the actual time values are provided at the corresponding experiment results sections.

7. RESULTS

We explored several different networks with different bit widths and clock speeds. The tables below will show the large differences between different network speeds. Furthermore, we performed one set of experiments with a buffer inserted between the CPU and the network to eliminate the possible jitter on the input of channel C_2, just as explained in Section 5. In the second set of experiments, we omitted the buffer.

7.1 Experiments with Buffer

The buffer at the input of channel C_2 resynchronizes the possibly bursty –or at least jittery– stream from P_3 on the CPU (see Figure 5) to a purely periodic stream. Hence, the network inputs are fully specified, allowing us analyze the network *without* the iterations mentioned in Section 5. After the network is analyzed, we have the input stream of P_1, and we can analyze the CPU scheduling. This will not only yield the performance of process P_1 but also the output event stream of P_3, which is finally required for dimensioning the buffer [23].

We performed experiments with three different network speeds: $480Mbyte/s$, $300Mbyte/s$ and $240Mbyte/s$, corresponding to an average network utilization of $U_{\text{net}} = 92,82\%$, $74,26\%$, and $46,41\%$, respectively. We expect that the propagation jitter on channel C_1 increases with increasing network utilization. In effect, the input jitter of P_1 will increase, in turn increasing the output jitter (or burstiness) of P_3, finally resulting in increasing buffering requirements.

Detailed results are provided in the appendix. In each table, the first column contains the process or network channel identifier, the second column provides the input jitter. The actual process or packet response time is given in the third column, and the output jitter is provided in the fourth column. Note that P_1 has no output. For each experiment, we also recorded the number of iterations of the response time calculation (fifth column). Finally, we also recorded the number of *busy period windows* as defined by [17, 32] which have to be analyzed in the presence of bursts. They represent some measure of the dynamic complexity of the component's schedule.

Experiment 1: The network speed is $480Mbyte/s$. The packet communication times are $17,5\mu s$, $6,7\mu s$, and $2,15\mu s$ for a complete data packet –possibly consisting of several network packets (see Sec, 6.1)– on channel C_1, C_2, and C_2, respectively. The timing analysis results are shown in Table 4 in the appendix. Using the approach in [23], we can bound the buffer size. In a worst case situation, we need to store $\lceil \frac{265\mu s}{50\mu s} \rceil = 12$ events, each representing a $3kB$ data block, resulting in a $36kB$ buffer.

Experiment 2: The network speed is $300Mbyte/s$. The packet communication times are $27,95\mu s$, $10,72\mu s$, and $3,43\mu s$ on channel C_1, C_2, and C_2, respectively. For the results, see Table 5. The differences compared to Experiment 1 are in bold letters. We see, that the network output jitters have actually increased. Also the number of iterations for the response time calculation has increased. However, this has not yet affected the execution of the CPU, i. e. the response times are still the same as in experiment 1. So, also the buffering requirements remain constant. This will change in the next experiment.

Experiment 3: The network speed is reduced to $240 Mbyte/s$. The packet communication times are $34,93\mu s$, $13,4\mu s$, and $4,29\mu s$ on channel C_1, C_2, and C_2, respectively. For the results, see Table 6. Now, the output jitter of P_3 has increased, and the buffer is now required to store 13 events ($39kB$). Furthermore, 33 response time iterations [13 *busy-period windows*] were necessary compared to 14 [7] in the previous experiment.

7.2 Experiments without Buffer

In the second set of experiments, we omitted the buffer at the input of channel C_2. This has –as theoretically explained in Section 5– severe consequences for the overall analysis procedure. We now have to start the network analysis with an assumption on the not yet known output of P_3. We start by assuming a periodic stream with a frequency of $20kHz$ –just as in the experiments with buffer–, and analyze the network and the CPU. Then, we have to check the actual output of P_3 against our assumption. This process is iterated until the assumption is met, or –in case of the last experiment– the given deadline for packets on channel C_1 ($1ms$) is missed.

Experiment 4: The network speed is $480 Mbyte/s$, just as in Experiment 1. The results are shown in Table 7. In the final results after iteration 2, we can see that the bursty output of channel C_2 now heavily distorts the timing of channel C_1. In the previous experiments this was not the case because of the buffer. However, we observe no severe consequences for the overall system performance, mainly due to the very conservative over-dimensioning of the network. If we reduce the network performance as in Exp. 2 and 3, we expect notable changes.

Experiment 5: The network speed is now set to $300 Mbyte/s$, as in Experiment 2. The results are shown in Table 8. We can see that the iteration does not terminate before step 3. The jitters have increased but can still be bounded.

Experiment 6: In the final experiment, the network speed is set $250 Mbyte/s$, already very close to $240 Mbyte/s$ in Experiment 3. The packet communication times are $33,53\mu s$, $12,86\mu s$, and $4,12\mu s$ on channel C_1, C_2, and C_2, respectively. In Table 9, we see that the jitters on the channels constantly increase. This further leads to burst execution of P_1, and the process response times quickly increase. We can stop the iteration after step 3, since the deadline of channel C_1 is violated.

7.3 Result Interpretation

Table 3 gives an overview about the six experiments. We can see that the jitter on the lower priority channel C_1 gradually increases with decreasing network performance. However, the buffer at the input of C_2 resynchronizes these effects. So, the bus load is (compared to the experiments without buffer) relatively determinate.

When the buffer is removed, the system is still *stable* as long as the network performance is above a certain limit. Only we can see that the jitters increase much faster compared to experiments 1 to 3. If the network performance becomes too low, the jitters on both points of interest (C_1 output and C_2 input) seem to exponentially grow, and the deadline of channel C_1 is quickly violated (after the third iteration step).

Table 3: Results Overview

network speed [MByte/s]	network util [%]	buffer size [kB]	C_1 output jitter [μs]	C_2 input jitter [μs]	C_1 worst-case resp. [μs]
480	46,41	36	17,45	—	34,95
300	74,26	36	69,46	—	97,41
240	92,82	37	256,29	—	291,22
480	46,41	—	85,85	265	103,35
300	74,26	—	276,13	275	304,08
250	89,11	—	$> 987,82$	> 515	$> 1ms$

The results of the experiments show two things: First, our approach can be configured to analyze heterogeneous MpSoC designs without the need for highly specialized and complex formal models. All used formalisms are of similar complexity than the ones already widely accepted in industry. And secondly, the approach proved applicable and efficient. Especially the cyclic dependencies could be resolved without major convergence problems. This demonstrates the general validity of the approach.

8. CONCLUSION

The component integration step is critical in MpSoC design since it introduces complex component performance dependencies. For instance, the cyclic dependencies in Figure 5 can not be fully overseen by anyone in a design team. Finding simulation patterns covering all corner cases will soon become virtually impossible as MpSoCs grow in size and complexity, and performance verification is increasingly unreliable. In industry, there is an urgend need for formal performance verification support in MpSoC design.

We have seen that the host of work in formal real-time analysis can be nicely applied to individual, local components or subsystems. But the well established view on scheduling analysis has shown to be incompatible with the component integration style which is common practice in MpSoC design due to heavy component reuse. Not surprizingly, output event models and their importance in system-level analysis integration have been widely neglected, so far.

We saw that the existing input event models are incompatible with the event streams at component outputs, thereby prohibiting the system-level composition of several local techniques. By slightly extending the most popular and comprehensible event models –without increasing their complexity at the same time– we could overcome these incompatibilities. The newly defined self-contained *six class model* appears as an efficient compromise between model simplicity and completeness.

Starting from these models, we then developed model transformations in order to satisfy the interfacing requirements. As a result, we obtained a configurable and compositional analysis procedure that is –in general– capable to analyze the performance of arbitrarily complex MpSoC designs. The transformations consider event stream propagation, event model interfacing, and iterations in cyclic systems *without* degrading model simplicity. Ultimately, this compositional approach can pave the way for a general application of real-time analysis techniques in MpSoC design.

9. REFERENCES

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Journal of Real-Time Systems*, 8(5):284–292, 1993.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems*, pages 133–137, 1991.

[4] AXYS Design. *MaxSim Development Suite*. http://www.axysdesign.com/products/products_maxsim.asp.

[5] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Journal of Real-Time Systems*, 24(1):93–128, 2003.

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

[7] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. Approximate schedulability analysis. In *Proceedings IEEE Real-Time Systems Symposium*, Austin, TX, USA, 2002.

[8] P. Chou and G. Borriello. Modal processes: Towards enhanced retargetability through control composition of distributed embedded systems. In *Proc. Design Automation Conference (DAC)*, pages 88–93, San Francisco, USA, June 1998.

[9] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor soc for advanced set-top box and digital tv systems. *IEEE Design & Test of Computers*, pages 21–31, Sip–Oct. 2001.

[10] ETAS, formerly Livedevices. *Real-Time Architect*. http://www.livedevices.co.uk/realtime.shtml.

[11] R. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Appl. Math., 1969.*, 17:416–429, March 1969.

[12] 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.

[13] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proceedings Real-Time Systems Symposiom*, Phoenix, Arizona, 1999.

[14] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1999.

[15] 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.

[16] Hermann Kopetz. *Real-Time Systems – Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Massachusetts, april 1997.

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

[18] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings Real-Time Systems Symposiom*, pages 166–171, IEEE Computer Society Press, 1989.

[19] 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.

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

[21] 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.

[22] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. In *Proc. Design, Automation and Test in Europe (DATE 2003)*, Munich, Germany, 2003.

[23] 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.

[24] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceeding 39th Design Automation Conference*, New Orleans, USA, June 2002.

[25] Semiconductor Industry Association. *2001 International Technology Roadmap for Semiconductors*. http://public.itrs.net/Files/2001ITRS/Home.htm.

[26] Sonics. *SiliconBackplane μNetwork*. http://www.sonicsinc.com/Pages/Networks.html.

[27] 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.

[28] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. *DEADLINE SCHEDULING FOR REAL-TIME SYSTEMS – EDF and Related Algorithms*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1998.

[29] Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings International Symposium on Circuits and Systems (ISCAS)*, Geneva, Switzerland, 2000.

[30] TimeSys. *TimeSys*. http://www.timesys.com/index.cfm?bdy=tools_bdy_model.cfm.

[31] K. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS 221, Department of Computer Science, University of York, UK, 1994.

[32] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems*, 6(2):133–152, Mar 1994.

[33] K. Tindell and J. Clark. Holistic schedulability analysis for distributed real-time systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[34] Tri-Pacific Software, Inc. *RAPID RMA*. http://www.tripac.com/html/prod-fact-rrm.html.

[35] TTTech AG. *TTP - Time-Triggered Protocol*. http://www.tttech.com/.

Appendix A — Tables

Table 4: Experiment 1

ID	input jitter [μs]	resp. time [μs]	output jitter [μs]	iter-ations	# of windows
C_1	0	[17,5;34,95]	17,45	3	1
C_2	0	[6,7;11,55]	4,85	3	1
C_3	0	[2,15;2,70]	0,55	1	1
P_1	17,45	[250;265]	—	1	1
P_3	0	[10;275]	265	14	7

Table 5: Experiment 2

ID	input jitter [μs]	resp. time [μs]	output jitter [μs]	iter-ations	# of windows
C_1	0	[27,95;97,41]	**69,46**	**6**	1
C_2	0	[10,72;25,31]	**14,59**	**4**	1
C_3	0	[3,43;4,3]	**0,87**	1	1
P_1	**69,46**	[250;265]	—	1	1
P_3	0	[10;275]	265	14	7

Table 6: Experiment 3

ID	input jitter [μs]	resp. time [μs]	output jitter [μs]	iter-ations	# of windows
C_1	0	[34,93;291,22]	**256,29**	**16**	1
C_2	0	[13,4;40,24]	**26,84**	**5**	1
C_3	0	[4,29,5,39]	**1,10**	1	1
P_1	**256,29**	[250;265]	—	1	1
P_3	0	**[10;285]**	**275**	**33**	**13**

Table 7: Experiment 4

ID	input jitter [μs]	resp. time [μs]	output jitter [μs]	iter-ations	# of windows
Iteration 1					
C_1	0	[17,5;34,95]	17,45	3	1
C_2	0	[6,7;11,55]	4,85	3	1
C_3	0	[2,15;2,70]	0,55	1	1
P_1	17,45	[250;265]	—	1	1
P_3	0	[10;275]	265	14	7
Iteration 2					
C_1	0	[17,5;**103,35**]	**85,85**	**7**	1
C_2	**265**	[6,7;11,55]	4,85	3	1
C_3	0	[2,15;2,70]	0,55	1	1
P_1	**85,85**	[250;265]	—	1	1
P_3	0	[10;275]	265	14	7
Iteration terminates after this step!!!					

Table 8: Experiment 5

ID	input jitter [μs]	resp. time [μs]	output jitter [μs]	iter-ations	# of windows
Iteration 1					
C_1	0	[27,95;97,41]	69,46	6	1
C_2	0	[10,72;25,31]	14,59	4	1
C_3	0	[3,43;4,3]	0,87	1	1
P_1	69,46	[250;265]	—	1	1
P_3	0	[10;275]	265	14	7
Iteration 2					
C_1	0	[27,95;**283,07**]	**255,12**	**10**	1
C_2	**265**	[10,72;25,31]	14,59	4	1
C_3	0	[3,43;4,3]	0,87	1	1
P_1	**255,12**	[250;265]	—	1	1
P_3	0	[10;**285**]	**275**	**33**	**13**
Iteration 3					
C_1	0	[27,95;**304,08**]	**276,13**	**11**	1
C_2	**275**	[10,72;25,31]	14,59	4	1
C_3	0	[3,43;4,3]	0,87	1	1
P_1	**276,13**	[250;265]	—	1	1
P_3	0	[10;285]	275	33	13
Iteration terminates after this step!!!					

Table 9: Experiment 6

ID	input jitter [μs]	resp. time [μs]	output jitter [μs]	iter-ations	# of windows
C_1	0	[33,53;233,79]	200,26	13	1
C_2	0	[12,86;34,51]	22,65	4	1
C_3	0	[4,12;5,17]	1,05	1	1
P_1	200,26	[250;265]	—	1	1
P_3	0	[10;275]	265	14	7
Iteration 2					
C_1	0	[33,53;**619,33**]	**585,8**	**18**	1
C_2	**265**	[12,86;34,51]	22,65	4	1
C_3	0	[4,12;5,17]	1,05	1	1
P_1	**585,8**	[250;**512,7**]	—	1	1
P_3	0	[10;**525**]	**515**	**53**	**20**
Iteration 3					
C_1	0	[33,53;***1021,35***]	**987,82**	**21**	1
C_2	**515**	[12,86;34,51]	22,65	4	1
C_3	0	[4,12;5,17]	1,05	1	1
P_1	**987,82**	[250;**1314,4**]	—	1	1
P_3	0	[10;**985**]	**975**	>**100**	>**30**
Termination because of deadline violation on channel C_1.					