# A Formal Approach to Performance Verification of Heterogeneous Architectures

Kai Richter, Rolf Ernst
Institute of Computer and Communication Network Engineering
Technical University of Braunschweig
Hans-Sommer-Strasse 66
D-38106 Braunschweig
{richter,ernst}@ida.ing.tu-bs.de

***Abstract***

*Performance and timing verification is critical for many embedded systems. For small systems, corner-case simulation patterns can be manually determined and timed simulation provides reliable system performance data. Larger heterogeneous multiprocessor systems with different operating systems and bus protocols exhibit complex scheduling anomalies which can not be fully overseen by anyone in a design team. In effect, simulation will likely miss critical corner cases. Such coverage problems have already lead to a variety of design errors in practice including transient overload, buffer overflow, and missed deadlines, and a more systematic approach is urgently needed. Semi-formal verification techniques are available for many practically used preemptive and non-preemptive hardware and software scheduling algorithms of processors and buses. However, they cannot be used in system-level analysis due to incompatibilities of their underlying event models. This paper presents a technique to couple the analysis of local scheduling strategies via an event interface model. We derive transformation rules between the most important event models and provide proofs where necessary. We use an expressive example to illustrate their application.*

## 1. Introduction

With increasing embedded system complexity, there is a trend towards heterogeneous architectures. Automotive systems include different processors running distributed functions with reactive or transformative behavior under the OSEK/VDX operating system standards with static or dynamic priority scheduling. Buses and networks use time sharing or packet communication. Similarly, multimedia devices run telecommunication protocols, coding and signal processing functions on heterogeneous VLSI-multiprocessors and coprocessors with several heterogeneous operating systems. Communication infrastructure migrates from simple busses to complex packet switched networks.

In all cases, system constraints, system specialization, and reuse of hardware and software IP (intellectual property) are the main sources of heterogeneity. Component specialization is needed to optimize system performance at low power and competitive cost. And using IP library elements in a copy&paste design style is the only way to reach the necessary design productivity.

Systems integration is a major challenge, since complex hardware and software component interactions pose a serious threat to all kinds of performance pitfalls including transient overloads, buffer overflows, and missed deadlines. The *International Technology Roadmap for Semiconductors* names system-level performance verification as one of the top-three design issues.

Timed simulation is state of the art in system performance verification. Tools such as MentorGraphic's *SeamlessCVE* [Seamless] and Axys Design Automation's *MaxSim* [MaxSim] support cycle-accurate cosimulation of the complete hardware/software system. The cosimulation times are extensive, but developers can use the same simulation environment, simulation patterns, and benchmarks in both function and performance verification. Simulation-

based performance verification, however, has *conceptual* disadvantages that become disabling as complexity increases.

Hardware and software component integration involves resource sharing that results in a confusing variety of performance dependencies at runtime which are not reflected in the system function. As an example, Figure 1 shows a CPU subsystem with three processes. Although all processes are activated periodically, the execution sequence is rather complex and leads to transient output bursts for $P_3$, modulated by $P_1$ execution.

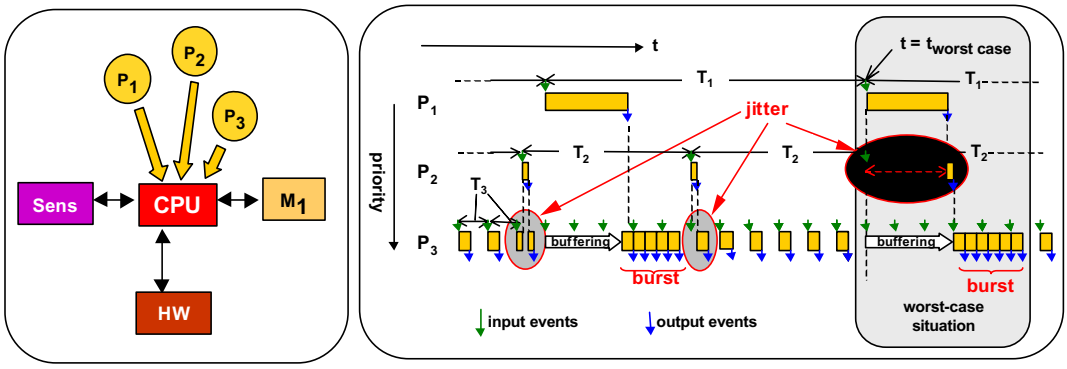


**Figure 1 Performance Dependencies due to Resource Sharing**

Component integration adds additional performance dependencies which can turn component best-case performance into system worst-case performance, a so called scheduling anomaly, as shown in Fig. 2. During bursts, the minimum (best-case) execution time of $P_3$ leads to maximum (worst-case) bus load, possibly slowing down other components communication. Such transient runtime effects represent critical execution scenarios, or corner cases, which have to be verified. However, such architectural corner cases can be subtle and extremely difficult to find and debug.

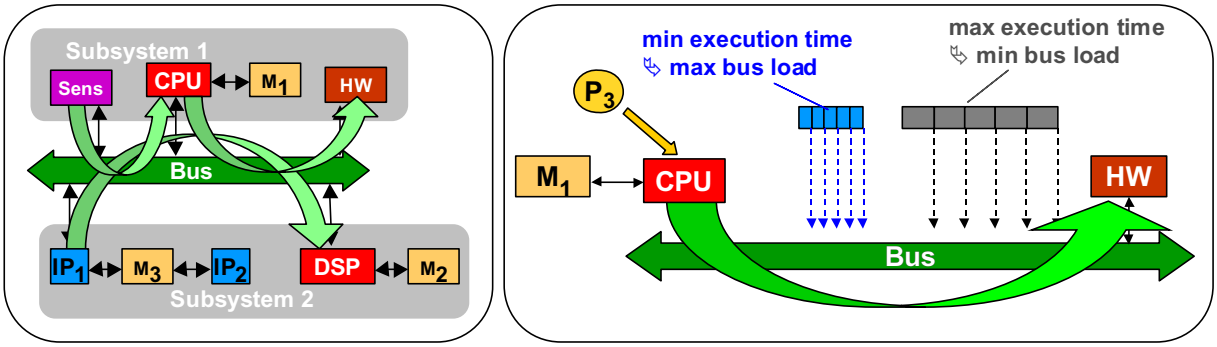


**Figure 2 Scheduling Anomaly due to Component Integration**

Where do we get the stimuli to cover all corner cases? Reusing function verification patterns is not sufficient because they do not cover the complex non-functional performance dependencies that resource sharing introduces (Fig. 1). Reusing component or subsystem verification patterns is not sufficient, either, since they do not consider the complex component and subsystem interactions (Fig. 2). The designer might be able to develop additional simulation patterns for simple systems with well understood component behavior. But manual corner case identification is not practical for complex multiprocessor systems with layered software architectures, dynamic communication protocols, and operating systems. In short, simulation-based performance verification is about to run out of steam and should be enhanced by formal approaches that inherently cover the corner cases.

## 2. Existing Formal Approaches to Performance Verification

### 2.1 Single-Component Techniques

Real-time systems research has addressed software scheduling analysis for decades, and many popular techniques are available. Examples include rate-monotonic scheduling [RMS] and earliest deadline first [EDF] using both, static and dynamic priorities; time-slicing mechanisms like time-division multiple access (TDMA) or round-robin (RR); and static order scheduling of synchronous data-flow [SDF]. Many extensions have found their way into commercial analysis and optimization tools such as Livedevices' *Real-Time Architect* [RTA], TriPacific *RapidRMA* [RapidRMA], and many more.

These techniques rely on a simple yet powerful abstraction of task and communication activation. Instead of considering each event individually, as simulation does, formal scheduling analysis abstracts from individual events to *event streams*. The analysis requires only a few simple characteristics of event streams, such as an event period or a maximum allowed jitter. From these parameters, the analysis *systematically* derives *corner-case scheduling scenarios* (an example is shown in Fig. 1), which safely bound the process and communication *response times*.

We just mentioned that the local techniques assume certain input event models, such as, e.g., periodic events with a bounded jitter. When integrating several components, the output of one component becomes the input of a connected component. Interestingly, output event models have been widely neglected in formal real-time systems research, so far.

Each event experiences a delay when traveling through a component. Due to resource sharing (and other influences), these delays can vary from one execution to the next, reflected by a response time *interval* that introduces uncertainty to the output events timing. In other words, each component adds jitter characteristics to the event stream, as already shown in Figure 1. Accumulating jitters can further lead to heavy event bursts. It is quite obvious that output event models are usually more complex than input event models, as can be seen when comparing the bus input and output event models in Figure 3.

Unfortunately, most output event models are not supported as input models by the known analysis approaches. Hence, the local techniques cannot be reasonably combined into a system-level analysis. They are limited to a single scheduling strategy, but fail to consider systems with multiple resource-sharing strategies and complex component interactions because of model incompatibilities, indicated by the gray boxes in Figure 3. The DSP, for instance, requires strictly periodic input to efficiently run a set of signal processing applications. For the HW component, we might only know a maximum allowed frequency which corresponds to a minimum time separation of two consecutive executions. Real-time systems research captures such minimum *interarrival times* using the model of *sporadic events*.

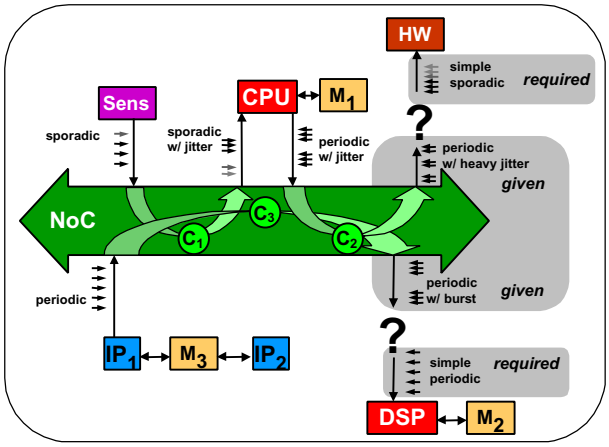


**Figure 3 Input and Output Event Models**

### 2.2. System-Level Techniques

"Holistic" analysis aims at finding timing equations capturing *all* influences and dependencies for an entire system. However, the system-level timing equations can be complex, and there is –so far– no general procedure to efficiently create and solve them. The known approaches are limited to specialized classes of distributed systems [Tindell, Eles, EDF] which simplifies the timing equations. For instance, the TTP (time-triggered protocol) provides a global synchronization between all processors over the TTP-bus. The obtained predictability, however, comes at a significant performance price, a price that grows with system size and complexity. The static nature of TTP and other "highly predictable" protocols increases buffer sizing requirements, along with response times. In addition, such protocols do not adapt to dynamically changing load situations that are typical for reactive embedded systems. Holistic analysis and the conservative approach will therefore not scale well to future distributed multiprocessor platforms with multiple network protocols.

Other techniques rely on combining several local analysis techniques and circumvent the incompatibility problem mentioned at the end of Section 2.1 by using complex *generalized* event stream representations. Thiele et. al. [Thiele] define numerical upper- and lower-bound event arrival curves and use network calculus for analyzing the components. Gresser [Gresser] defines a complex event vector system, and component analysis is performed using event dependency matrices. However, both approaches define new models and require new scheduling analysis techniques, so they cannot reuse existing work, which we consider a major disadvantage for a broad industrial acceptance. We can summarize that there is currently no sufficiently optimal analysis approach that covers all aspects of todays heterogeneous architectures.

## 3. Our Basic Idea

We do not necessarily need to develop new analysis techniques or introduce new models, if we can benefit from the host of work in real-time scheduling analysis. We have recently developed a technology [Richter02, Richter03a] that lets us a) extract key output event stream information from a given component schedule, and b) automatically interface or adapt the output event stream to meet the established input event models such as RMS. This way, we overcome the mentioned model incompatibilities and designers and analysts can safely apply existing subsystem analysis techniques and tools without compromising component integration and system-level analysis.

### 3.1 Event Model Interfaces

With respect to the example in Figure 3, we introduce an *Event Model Interface* (EMIF) to transition from the model of periodic events with jitter to the model of sporadic events at the input of the HW component in Figure 4. Some analysis techniques need this transformation, requiring only a minimum of math. The jitter events are characterized by a period ($T_{C2}$) and a jitter ($J_{C2}$). The jitter bounds the maximum time deviation of each event with respect to a virtual reference period. So, each individual event can be at most $J_{C2}/2$ earlier or later than the reference. The required sporadic model has only one parameter, the minimum interarrival time ($t_{HW,min}$) between any two successive events, thus bounding the maximum transient event frequency. Now, imagine two successive jitter events, the first being as late as possible ($t_1=t_0+J_{C2}/2$) and the second as early as possible ($t_2=t_0+T_{C2}+J_{C2}/2$). The minimum distance between any two events is thus $t_{HW,min} = t_2 - t_1 = T_{C2} - J_{C2}$. EMIFs only change the mathematical event stream representation for analysis purposes. The system implementation and the actual timing of stream events remain unchanged.

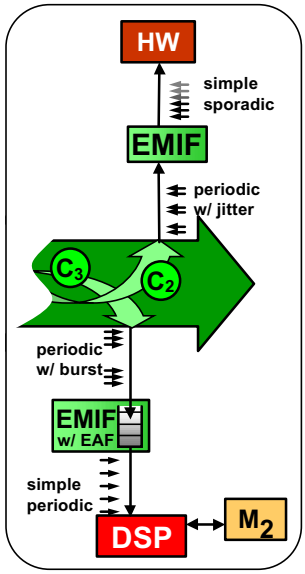


**Figure 4 Usage of EMIF and EAF**

### 3.2 Event Stream Adaptation

Such direct model transformations are not always possible. For instance, consider the periodic stream with burst entering the DSP in Figure 4. Many signal processing applications require purely periodic execution to run efficient DSP schedules. But the burst obviously has to be resynchronized to meet the required model of purely periodic events, i.e. the actual timing of individual events has to be changed. In such situations our technology automatically inserts an *Event Adaptation Function* (EAF) as a supplement to the EMIF to make the streams match. EAFs correspond to buffers that have to be inserted at a component interface. Hence, EAFs modify the system to make it working *and* analyzable. Optimized buffer sizing and buffering delay calculation is automatically performed during adaptation. The math for the resulting EMIF in the example is –again– relatively simple. The sought-after parameter ($T_{DSP}$) of the periodic stream is the average period of the burst stream, and can be described by the burst's *outer period* ($T_{C3}$) and its *burst length* ($n_{C3}$): $T_{DSP}=T_{C3}/n_{C3}$.

### 3.3 Event Propagation and Analysis Principle

The EMIF and EAF technology presented in the preceding sections allows to transition between a variety of different event models for a single event stream, thereby overcoming the mentioned model incompatibilities. This is then used to map the output event models of components to the input models of the connected components. As a result, each local component analysis can use the model best suited for a particular component. We call this "scheduling analysis integration" [Richter03b].

The overall system-level analysis principle is shown in Figure 5. First, the known environmental timing *assertions* are applied to the primary system inputs, i.e. those components that directly communicate with the system environment. This applies to environmental data and event sources such as network interfaces or sensor devices. Then, these components are analyzed using known techniques providing common results such as component utilization (percentage of available processing power or bus bandwidth), response times for tasks and communication needed to check system latency constraints, local buffer memory requirements, etc. Finally, the output event models are determined and are mapped to the input models of the connected components, possibly using EMIFs and EAFs, just as explained above. This process is iterated until all components are analyzed.

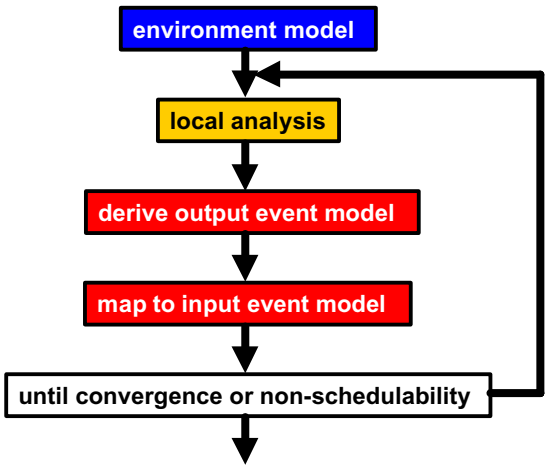


**Figure 5 Analysis Principle**

In pure feed-forward systems, i.e. systems without cycles, the procedure is relative simple. The environmental input event models are just propagated through all local component analyses to the system outputs. Finally, global input-output delays and global buffer size requirements can be determined.

More complex iterations can be found in systems where the timing of two or more components is mutually dependent. Feed-back communication is often found in recursive filter applications in signal processing, representing a *functional* dependency cycle. However, system integration can also lead to *non-functional* mutual dependencies which are far less intuitive. Figure 6 highlights a nonfunctional event stream dependency cycle in our example system that is only introduced by communication sharing. Upon receipt of new sensor data, the CPU activates process P1, which preempts P3 and thus affects the execution timing of P3 . Figure 1 illustrates this preemption. P3 ’s output, in turn, enters the network on channel C2 , where it now interferes with the arriving sensor data on C1. The interference of the two functionally independent channels, C1 and C2, closes the dependency cycle. Note that the subsystem in Figure 1 was originally cycle-free.

Such cycles are analyzed by iterative propagation of event streams until the event stream parameters converge or until a process misses a deadline or exceeds a buffer limit. This iteration process terminates because the event timing uncertainty, i.e. the best-case to worst-case event timing interval, grows monotonically with every iteration. For cases in which no convergence occurs automatically, we have developed a mechanism that uses EAFs to break up the dependency cycle and enforce convergence by reducing the timing uncertainty. We have thoroughly investigated cyclic dependencies. Note that the event flow cycles are not an artificial result of global analysis but exist in practice as the example demonstrates. And the event stream view allows to optimize buffer sizing in such situations which are usually very hard to find using simulation.

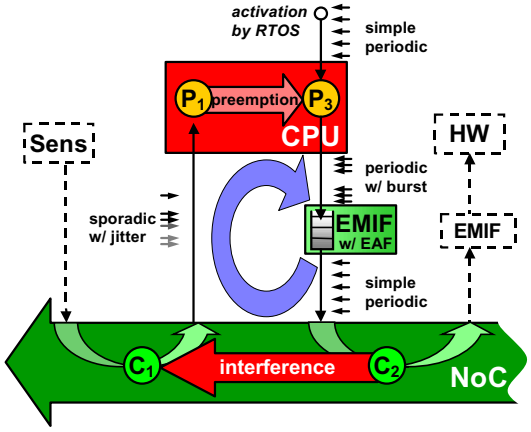


**Figure 6 Non-Functional Event Stream Dependency Cycle**

## 4. Experiments

So far, we presented event model propagation, interfacing, and adaptation. Now, we will demonstrate the applicability of the approach by fully analyzing the example system.

### 4.1. Set-Up

#### The Actuators

There are three actuators: The sensor sporadically sends data blocks of *8kb* size to P1, with a maximum sending frequency of *1,7kHz*, which corresponds to a *sporadic event model* with a minimum sporadic period *588µs*. Process P3 is periodically activated by the RTOS (real-time operating system) on the CPU with a period of *50µs*. The high-performance DSP application on IP1 has a sending frequency of *140kHz*, corresponding to a period of *7,14µs*.

#### 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 P3 are split into 24 packets of *(128 +6) = 134bytes*. This channel C2 has a higher priority than channel C1. The highest-priority channel C3 does not split the DSP data packets, but only adds the *6byte* protocol information. The overall average network load is *222,77Mbyte/s.*

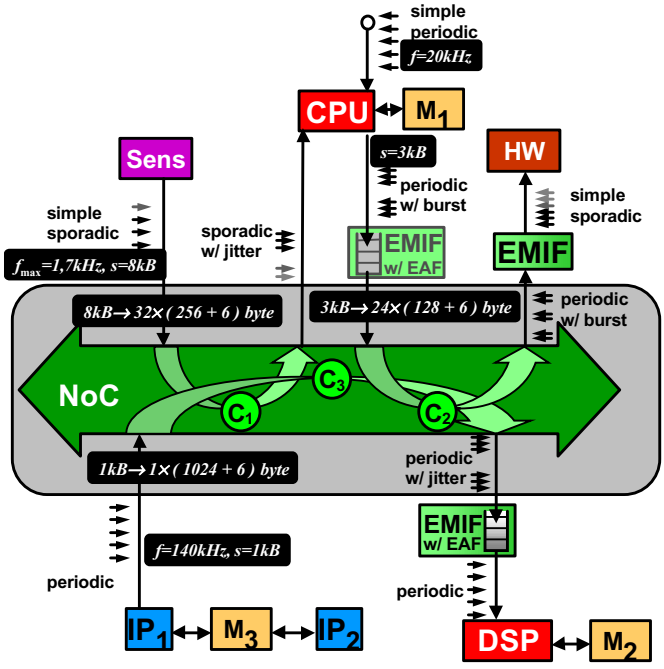


**Figure 7 Example System with Timing Set-Up**

**Execution and Transmission Times**

For simplicity, the *core execution times* of the two processes on the CPU are assumed constant: *250µs* for P1 and *10µs* for P3. This is not a limitation of the approach but rather a clarification of the following experiments.

The transmission 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.

We explored several different networks with different bit widths and clock speeds. Furthermore, we performed one set of experiments with a buffer inserted between the CPU and the network to eliminate the possible jitter or burst on the input of channel C2, just as explained in Section 3.3. In the second set of experiments, we omitted the buffer.

### 4.2 Experiments with Buffer

The buffer at the input of channel C2 resynchronizes the possibly bursty –or at least jittery– stream from P3 on the CPU (see Figure 6) to a purely periodic stream. Hence, the network inputs are fully specified, allowing us to analyze the network *without* the iterations mentioned in Section 3.3. After the network is analyzed, we have the input stream of P1, and we can analyze the CPU scheduling. This will not only yield the performance of process P1 but also the output event stream of P3, which is finally required for dimensioning the buffer. We performed experiments with three different network speeds: *480Mbyte*/*s*, *300Mbyte*/*s* and *240Mbyte*/*s*, corresponding to an average network utilization of 46,41%, 74,26%, and 92,82%, respectively.

We expect that the propagation jitter on channel C1 increases with increasing network utilization. In effect, the input jitter of P1 will increase, in turn increasing the output jitter (or burstiness) of P3, finally resulting in increasing buffering requirements.

**Experiment 1:** The network speed is *480Mbyte*/*s*. The packet transmission times are *17,5µs*, *6,7µs*, and *2,15µs* for a complete data packet on channel C1, C2, and C2, respectively. The output jitters are *4,85µs* for C2 and *265µs* for P3. In a worst case situation, we need to store 6 events, each representing a *3kB* data block, resulting in a *18kB* buffer. The C1 output jitter is *17,45µs.*

**Experiment 2:** The network speed is *300Mbyte*/*s*. The packet transmission times are *27,95µs*, *10,72µs*, and *3,43µs* on channel C1, C2, and C2, respectively. The network output jitter of C2 increases to *14,59µs*, but the buffering requirements remain constant. Only the C1 output jitter increases to *69,5µs.*

**Experiment 3:** The network speed is reduced *to 240Mbyte*/*s*. The packet transmission times are *34,93µ*s, *13,4µ*s, and *4,29µs.* Now, the output jitter of P3 has increased to *275µs*, the buffer is now required to store 7 events (*21kB*), and the output jitter of C1 further increases to *256,3µs.*

### 4.3 Experiments without Buffer

In the second set of experiments, we omitted the buffer at the input of channel C2 . This has –as theoretically explained in Section 3.3– 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 P3 . 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 P3 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 C1 (*1m*s) is missed.

**Experiment 4:** The network speed is *480Mbyte*/*s*, just as in Experiment 1. The output jitters of C2 and P3 remain unchanged compared to experiment 1. However, since the jittered output of P3 now enters the bus without synchronization, C1 communication is heavily distorted, so the

C1 output jitter is *85,9µs*. 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 *300Mbyte/s*, as in Experiment 2. Now, the output jitters increase: *14,59µs* for C2, *275µs* for P3. Again, C1 communication is further distorted and the output jitter increases to *276,13µs*, exceeding the value of Experiment 3.

**Experiment 6:** In the final experiment, the network speed is set *250Mbyte/s*, already very close to *240Mbyte/s* in Experiment 3. The jitters on the channels constantly increase during the iterative event stream propagation, further leading to heavy burst execution of P1. After the third iteration, the response time of C1 exceeds the given deadline of *1ms* and we can stop the iteration. The system is not schedulable.

### 4.4 Result Interpretation

| network speed *[Mbyte/s]* | network util *[%]* | buffer size *[kb]* | C1 output jitter *[µs]* | C2 input jitter *[µs]* | C1 worst-case response *[µs]* |
|---|---|---|---|---|---|
| *480* | *46,4* | *18* | *17,4* | *-* | *34,9* |
| *300* | *74,3* | *18* | *69,5* | *-* | *97,4* |
| *240* | *92,9* | *21* | *256,3* | *-* | *291,2* |
| *480* | *46,4* | *-* | *85,9* | *265* | *103,4* |
| *300* | *74,3* | *-* | *276,1* | *275* | *304,1* |
| *250* | *89,1* | *-* | *> 980* | *> 515* | *> 1ms* |

**Table 1 Result Overview**

Table 1 gives an overview about the six experiments. We can see that the jitter on the lower priority channel C1 gradually increases with decreasing network performance. However, the buffer at the input of C2 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 (C1 output and C2 input) seem to exponentially grow, and the deadline of channel C1 is quickly violated (after the third iteration step).

The results of the experiments show two things: First, our approach can be configured to analyze heterogeneous 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, e.g. [RMS]. And secondly, the approach proved applicable and efficient. Especially the cyclic dependencies could be resolved without major convergence problems. On an 2.4 GHz Pentium P4 CPU, the runtime of our analysis tool SymTA/S (see next Section) is below 1 second (!) for each of the given experiments.

## 5. The SymTA/S Tool

We have developed a Java-based tool prototype that we call SymTA/S (Symbolic Timing Analysis for Systems). We have a similar tool SymTA/P (SymTA for Processes) to determine the core execution times of tasks.

An easy-to-use GUI allows to configure the analysis in SymTA/S. Figure 8 shows a screenshot of the tool for the example system of Figure 7. The user tasks and communication channels are edited and connected in the main "drawing area". The environmental assertions and constraints are modeled as "virtual" source and sink tasks. The mapping of tasks and communications to resources is shown in the "architecture and mapping window".

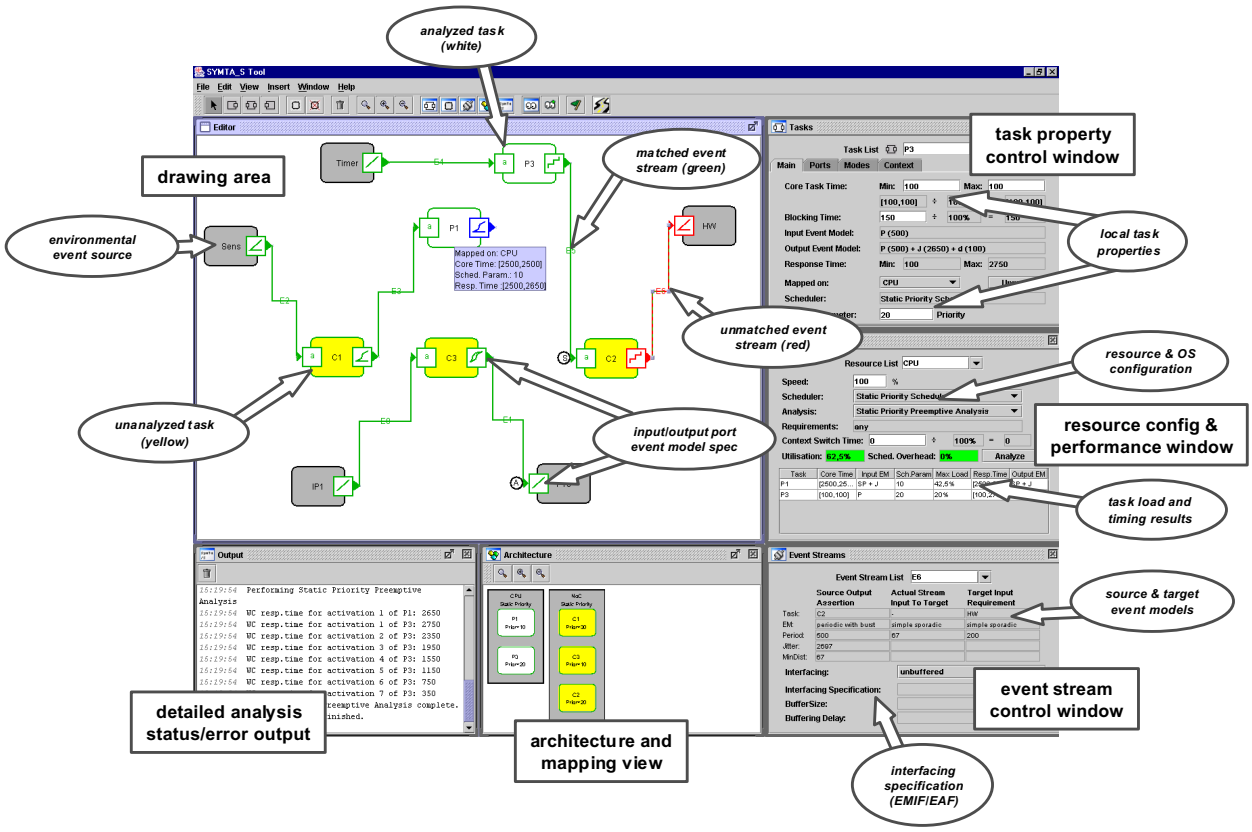


**Figure 8 SymTA/S Tool Screenshot**

Task parameters such as the core execution time and scheduling parameters are input in the "task property window" in the top right corner of the tool. The "resource config window" in the middle right contains the operating system configuration, scheduling strategy, OS overhead, etc.. The event streams representing the component interactions are observed in the "event stream window" in the lower right corner, where also the interfacing specifications such as EMIF/EAF are described and optimized. The tables in the three windows on the right hold all timing information about the entire system and are continuously updated during analysis.

The actual analysis is started by a simple "double-click" mechanism. The mathematical background on event model interfacing and adaptation is fully hidden from the user, so she/he can concentrate on the key tasks, that is integration and analysis. In order to allow fast and easy tool control, key information about the analysis status (schedulable or not) and the up-to-dateness of results in an iterative analysis process are visualized using an intuitive color code: green=success, red=failure, white=up-to-date, yellow=needs update, etc.. The timing parameters of central concern such as response times, buffer sizes, and jitters can optionally be shown as tool tips when the mouse is moved over a specific element in the drawing area.

## 5. Conclusion

The current design trend of component reuse and the "copy&paste" integration style leads to increasingly heterogeneous embedded system architectures, posing a serious threat to all kinds of performance pitfalls. Design errors such as transient overload and missed deadlines can be subtle and difficult to detect and to debug using the traditional performance simulation techniques, and formal approaches are becoming attractive as a reasonable supplement. Systematic scheduling analysis techniques are known and practically applied to individual components for a long time. But so far, incompatibilities in the underlying load and scheduling models prevent their direct application in heterogeneous system design.

Our technology generally supports automatic interfacing and adaptation for the most frequently used event models, allowing designers to apply known analysis and optimization techniques locally without compromising global analysis. The elegant event stream and interfacing view helps designers to understand and control the complex component interactions and to optimize the dynamic behavior of the overall system, effectively enabling fast and reliable system integration.

The event model interfaces and adaptation functions enable a novel system-level analysis of complex heterogeneous systems. We consider our approach a serious alternative or supplement to simulation-based performance verification. It allows comprehensive system integration and provides much more reliable performance measures at far less computation time.

We have already applied out technology to case studies in cooperation with our industry partners in telecommunications, multimedia, and automotive manufacturing. Each case had a very different focus. In the telecommunications project, we resolved a severe transient-fault system integration problem that not even prototyping could solve. In the multimedia case study, we modeled and analyzed a complex two-stage dynamic memory scheduler to derive maximum response times for buffer sizing and priority assignment. In the automotive study, we showed how the technology enables a formal software certification procedure.

**References**


[Seamless] "Seamless Hardware/Software Co-Verification" http://www.mentor.com/seamless/, Mentor Graphics Inc.

[MaxSim] "MaxSim Developer Suite" http://www.axysdesign.com/products/products_maxsim.asp, Axys GmbH

[RMS] "Scheduling algorithms for multiprogramming in a hard-real-time environment", C. L. Liu and J. W. Layland, Journal of the ACM, Vol. 20, No. 1, 1973

[EDF] "DEADLINE SCHEDULING FOR REAL-TIME SYSTEMS -- EDF and Related Algorithms", J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, Kluver Academic Publishers, 1998

[SDF] "Synchronous Dataflow", E. A. Lee and D. G. Messerschmitt, Proceedings of the IEEE, Vol. 75, No. 9, 1987

[RTA] "Real-Time Architect", http://www.livedevices.com/realtime.shtml, ETAS GmbH, formerly Livedevices Inc.

[RapidRMA] "RAPID RMA: The Art of Modeling Real-Time Systems", http://www.tripac.com/html/prod-fact-rrm.html, TriPacific Software Inc.

[Tindell] "Holistic Schedulability Analysis for Distributed Real-Time Systems", K. Tindell and J. Clark, Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems), Vol. 40, 1994

[Eles] "Bus Access Optimization for Distributed Embedded Systems Based on Schedulability Analysis", P. Pop, P. Eles, and Z. Peng, Proc. Design, Automation and Test in Europe (DATE) Conference, 2000

[Thiele] "Real-time Calculus for Scheduling Hard Real-Time Systems", Lothar Thiele, Samarjit Chakraborty and Martin Naedele, Proceedings International Symposium on Circuits and Systems (ISCAS), 2000

[Gresser] "An Event Model for Deadline Verification of Hard Real-Time Systems", K. Gresser, Proceedings 5th Euromicro Workshop on Real-Time Systems, 1993

[Richter02] "Event Model Interfaces for Heterogeneous System Analysis", K. Richter and R. Ernst, Proc. Design Automation and Test in Europe (DATE) Conference, 2002

[Richter03a] "A Formal Approach to MpSoC Performance Verification", K. Richter, M. Jersak, and Rolf Ernst, IEEE Computer Magazine, April 2003

[Richter03b] "Scheduling Analysis Integration for Heterogeneous Multiprocessor SoC", K. Richter, R. Racu, and Rolf Ernst, Proc. IEEE International Symposium on Real-Time Systems (RTSS), 2003