Breaking Down Complexity for Reliable System-Level Timing Validation

Dirk Ziegenbein, Marek Jersak, Kai Richter, and Rolf Ernst

Institute for Computer and Communication Network Engineering
Technical University of Braunschweig, Germany
{ziegenbein, richter, jersak, ernst}@ida.ing.tu-bs.de

Abstract

Complex embedded systems consist of hardware and software components from different domains, such as control and signal processing, many of them supplied by different internal and external source (e. g. IP). The system architect faces the challenge to integrate, optimize and validate the resulting heterogeneous systems. The analysis of the whole system is currently limited to simulation or emulation, since formal validation is only available for some subproblems. While simulation still seems viable for the validation of the system function, it can not be reliably applied to the validation of non-functional system properties, in particular timing. In this paper, we propose a validation methodology which augments existing cosimulation-based design approaches with formal timing analysis capabilities. This methodology is based on the *decomposition* of the validation task into the analysis of individual processes and resources for which formal analysis techniques are known and on the *composition* of the obtained results in order to obtain system-level timing information. Furthermore, it is shown how the analyzability of system timing can be systematically improved by slightly changing the system implementation.

1 Introduction

The complexity of embedded systems has steadily grown in recent years. Fueled by increasing device integration capabilities of silicon technology, more and more functions can be implemented on a single chip leading to systems-on-chip (SoC). Another dimension of the growing complexity is the resulting heterogeneity of application and architecture. Due to different functional characteristics (e. g., reactive or transformative), the SoC application is typically captured using several domain-specific languages with possibly fundamentally different underlying models of computation [3]. Typical SoC architectures consist of a combination of different processor types, specialized memories, and weakly programmable or dedicated HW components connected using complex on-chip networks consisting of buses, switches or point-to-point connections. This variety of different architectural components is a result of specialization and optimization which is necessary in order to achieve the required performance at low cost and low power consumption. Adding to this heterogeneity is the reuse of intellectual property (IP) components which is inevitable in order to reach the design productivity required to meet time-to-market constraints.

The main challenge in the design of these complex heterogeneous SoC is the reliable system validation in order to allow a safe integration of the different system parts. First, there is the well known problem of system function validation. System function validation determines if the implemented function equals the specified function (equivalence checking) or if certain system properties are met (model checking). There are formal function validation tools, but they are typically only applied to components, whereas simulation is the preferred means to system-level function validation. Here, simulation pattern development is the main challenge for designers. System function validation has been in the focus of EDA and software engineering for years, and there are many methods supporting different stages of the design process including executable specifications and rapid prototyping.

There is, however, a second validation problem concerned with the validation of non-functional system properties such as timing, required memory size, and power consumption. While the ideas presented in this paper are valid for the verification of non-functional properties in general, the focus of the paper is on timing which is typically most critical for correct system behavior. Traditionally, timing validation was mostly in the focus of safety critical or high availability systems where formal quality metrics were applied. In all other systems, simulation and prototyping were assumed to cover timing analysis as a side effect of system function validation. Specific timing analysis was at most used on an abstract level such as in statistical communication network analysis.

This situation is changing with increasing system complexity. Never before, technical systems with similar heterogeneity and complexity had to be built with a comparable productivity. Such a complex system, composed by a system-level "cut-and-paste" approach, exposes a confusing variety of communication and run-time interdependencies, which cannot be fully overseen by anyone in a design team. Both upper and lower run-time and communication bounds and their combinations contribute to this complex behavior. Decades of work in real-time operating systems has revealed run-time anomalies [4] which are not intuitive and lead to risks that are not known to most designers.

There are approaches to extend simulation to performance analysis of complex heterogeneous systems. Most prominent is VCC from Cadence [1]. VCC requires sufficient input patterns to cover all corner cases. Unfortunately, it is not clear how to combine component corner cases in order to obtain system corner cases. An example is a process generating maximum event bursts and bus loads at minimum execution time which is normally not of interest in component design. If the system integrator is aware of this particular problem, he/she might be able to add new simulation patterns, but only if the system is sufficiently simple. If we add the software architecture complexity with APIs, drivers, and operating system components, then it becomes obvious that system-level corner case identification is no practical option.

The current practice of reusing component corner cases in system design does, therefore, not scale to large systems. Hence, VCC and related simulation tools are inappropriate for the upcoming problems of complex system performance analysis. It must be emphasized that the situation is different for system function validation, where simulation still seems viable.

Alternative approaches using statistical or typical performance data are increasingly unsafe in their inability to detect memory overflow and transient overload leading to data dependent transient system errors which are extremely difficult to debug and threaten system quality.

In this context, formal analysis techniques are an appealing alternative for validation of non-functional system properties. In contrast to simulation, formal analysis ensures complete corner case coverage by nature. With a carefully selected level of system and component abstraction, the results are conservative, i. e. they are guaranteed under all circumstances. Formal analysis approaches

capture process and system properties using parameterized mathematical models. Depending on their intended use, such models account for instruction execution, critical program paths, scheduling influence, or process interaction and support the derivation of conservative bounds on process core execution times, system response times, or required memory size.

Unfortunately, system complexity and heterogeneity are major obstacles for the system-wide application of formal validation methods. As shown in Figure 1 and presented in an overview of existing work in Section 2, reliable validation of system timing is only available for systems with either a low application complexity (e. g., single process) or a low architectural complexity (e. g., single resource). However, Figure 1 also shows that todays heterogeneous SoC have a high complexity with respect to application as well as architecture. The lack of reliable timing validation for this class of systems has already lead to costly delays in product delivery. Recent popular examples can be found in automotive, mobile communication, and set-top box industries.

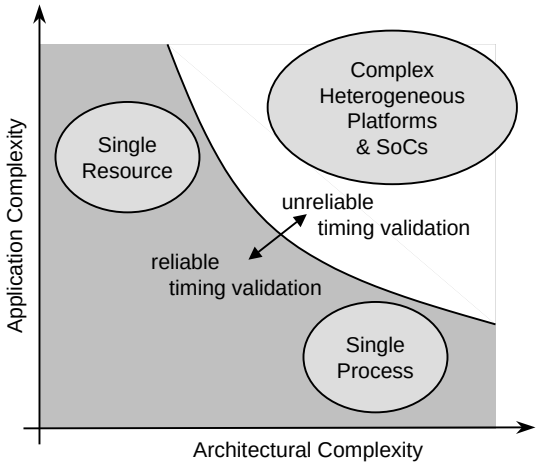

Figure 1: Availability of reliable timing validation methods with respect to application and architectural complexity

This paper proposes a validation methodology which tackles the problem of system timing validation and, in particular, addresses the following two critical questions:

- How can we break down the system complexity in order to use existing formal validation tools and techniques for system-level timing analysis of heterogeneous SoC?
- How can we implement systems in order to increase their analyzability?

The paper is organized as follows. After an overview of existing approaches to timing analysis on various levels of abstraction in Section 2, the proposed validation methodology is described in Section 3. This section first presents the basic ideas of our approach before combining them to a complete validation flow. Then, a systematic approach to adapt a system implementation for increased analyzability is informally presented in Section 4. The paper is concluded by a summary and an outlook on future work.

2 Related Work

Formal timing validation methods can be divided into process-level and system-level timing analysis. The former methods analyze a single process mapped to a resource of the target architecture and yield the core execution time of the process, whereas the latter methods analyze resource sharing effects and yield process response times for given core execution times.

For process-level timing analysis, the *sum-of-basic-blocks* model [12] is established as a standard approach. Here, the overall task execution time is the sum of all basic block execution times multiplied by the corresponding execution count for each of the basic blocks. Both values, time and count, are intervals representing the worst case and best case bounds. As a result, the overall execution time is an interval, too.

Many other approaches to task execution time analysis are also based on the analysis granularity of basic blocks or single basic block transitions [7] or require complex modifications to execution time determination [15]. Very few approaches like [8] also consider more fine-grain influences of complex processor architectures, e. g. pipelines and super-scalar machines. The major drawback of such detailed approaches is state-space explosion.

The SYMTA (SYMbolic Timing Analysis) tool suite [22] extends the sum-of-basic-blocks approach by raising the analysis granularity from basic blocks to task segments which are sequences of basic blocks having a single, input data independent control flow across basic block boundaries. Due to the raised granularity, the number of points where worst case assumptions (e. g., empty pipeline or cash flush) have to be made is reduced leading to tighter bounds.

Additionally, SYMTA allows to specify task execution contexts by providing information on input data that influences input data dependent control structures. This way, execution paths are selected and task segments can be merged even further. As a result, not only a single task execution time interval but also a set of comparatively narrow execution time intervals, one for each context, can be given.

For system-level timing analysis, there is a huge amount of work, mainly in the domain of real-time operating systems. These approaches capture system timing in a closed form using timing equations and appropriate solution algorithms which reflect the used resource sharing (scheduling) strategy. Example approaches include Liu and Layland who proposed a preemptive priority-driven scheduling to guarantee deadlines for periodic hard real-time processes [14]. They considered a static (Rate Monotonic) and a dynamic (Earliest Deadline First) priority assignment and provided a formal analysis framework for both. In [9], 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 allow periodic activation with jitter, e. g. [19], and arbitrary deadlines and burst [11] for static-priority scheduling. Sprunt et. al. [20] analyze the influence of sporadic process activation.

Unfortunately, all mentioned approaches assume a single coherent scheduling strategy for a given system, whether single or multi-processor. Very few exceptions consider special cases of more complex architectures, such as [16] analyzing response times for static priority process scheduling combined with a TDMA bus protocol. However, there is no general analysis approach capable of handling heterogeneity with respect to resources and scheduling strategies as required for complex SoC. Furthermore, it is doubtful that such a general approach providing a closed-form solution for arbitrarily complex system architectures can be found, mainly because of the highly complex dependencies of various influences on system-level timing in such systems. Rather, it seems to be more promising to systematically combine the existing process- and resource-level tools and techniques in order to obtain system-level timing parameters. Such a compositional approach is described in the following two sections.

3 Methodology

In this section, the proposed methodology for reliable timing validation of complex, heterogeneous SoC is presented. Please note that this methodology is not intended to be a complete methodology on its own but rather augments existing cosimulation-based methodologies with the capability to reliably validate system timing. This means that functional validation as well as synthesis is performed using established methods and tools from the cosimulation world (e. g., VCC [1] or CoWare [2]).

A major obstacle for a systematic system-level timing validation of complex SoC is their inherent architectural heterogeneity as well as the lack of coherency between the different languages used in a multi-language specification. Thus, a requirement for our methodology is a homogeneous model of the complete system which provides a coherent formal underpinning for the validation of system timing. This system model has to support the incremental back annotation of timing information to allow a stepwise analysis flow. Furthermore, the model should be capable of representing the system at various levels of detail in order to cope with complexity. These requirements are discussed in more detail in Section 3.1.

The proposed methodology assumes a given system implementation, i. e. the application has already been mapped to the target architecture and scheduling methods and parameters have been chosen. The architecture selection as well as the mapping and scheduling decisions result from a design exploration step, either manually, using existing cosimulation-based tools (e. g., VCC [1]), or based on heuristics or multi-objective optimization (e. g., [6]). Such explorations are usually based on rough performance estimates to quickly reject infeasible and non-optimal implementations out of the host of possibilities. Then, each of the remaining candidates is analyzed in detail using our formal methodology. Finally, such detailed results can be fed back to the exploration step in order to further reject candidates. As a result, only a small set of pareto-optimal [6] implementations will remain. As a side effect, the detailed performance analysis results help to improve the mentioned estimation techniques.

The basic idea of our methodology is to decompose the problem of system-level timing analysis in order to be able to use existing tools or approaches from industry and academia. The results of these tools are then combined in order to obtain system-level timing parameters such as end-to-end latencies. The decomposition divides the influences on system-level timing into two orthogonal classes. The first class is based on individual process analysis which assumes exclusive resource access for each process, whereas the second class is based on individual resource analysis which considers resource sharing influences. The decomposition is enabled by abstraction of the process interaction (for individual process analysis, see Section 3.2) and of the resource interaction (for individual resource analysis, see Section 3.3). The orthogonality of these concepts allows the composition of the obtained results. The validation flow of the proposed methodology is described in Section 3.4.

3.1 Abstract System Model

In this section, we informally introduce aspects of a system model which is well suited for system-level timing analysis. Such a system model has to satisfy two key requirements. On one hand, it is imperative to abstract all application and architecture details which are *not* necessary for system-level timing analysis. This allow to expose in a suitable form those system properties which are required for the application of formal analysis techniques. On the other hand, it must be possible to provide tight, yet conservative worst- and best-case bounds for all relevant properties. System-level timing analysis techniques have to consider intervals to be able to reliably validate timing [21].

Application System-level timing analysis is based on an operating system view of the application. Therefore, a suitable application representation are processes with input and output ports communicating via channels. No functional details have to be maintained. Instead, the process function is abstracted into a small set of properties. For processes activated by incoming data (or events, signals), the amount of data necessary for activation has to be captured. For time-driven processes, timer properties (e. g. period, jitter) have to be represented. Furthermore, the amount of communicated data per activation at each input/output port has to be captured to allow determining buffer loads and communication delays.

Architecture The architecture is reasonably represented by abstract processors, busses, and their interconnection structure. Processors are characterized by their type, clock speed, cache size and strategy, and busses by width and speed, respectively. Memory is usually associated with processors. In addition to this purely hardware-oriented view, operating systems including bus and other drivers are also part of the architecture. Here, the most important properties are scheduling strategies and bus (arbitration) protocols, including parameters like context switching time, header length, etc.

Mapping and Scheduling Mapping is the assignment of processes and channels to computation and communication resources, respectively. Scheduling decisions include the assignment of priorities (in case of a priority scheduler) or slot times (in case e. g. of a round-robin scheduler) for processes and channels.

Environment and Timing Constraints For system-level timing analysis, timing models for arriving data (or events, signals) from the environment are required. These provide information about activation frequency of processes at the system inputs. End-to-end latency constraints are required to enforce process execution.

The above requirements for an abstract system representation for system-level timing analysis are met by the recently introduced SPI (**S**ystem **P**roperty **I**ntervals) model [24], which we use as the formal basis for our work. The SPI model is the basis of the SPI workbench, an open, international research effort to enable integration of various tools and techniques for system-level modeling, exploration, analysis and validation of non-functional system properties.

3.2 Process Interaction Abstraction

We have identified two key elements to reduce application complexity. The first is to transform non-functional properties of an application into a representation which is independent of domain-specific input languages. This has two benefits: on one hand system-level timing analysis techniques that work on this representation are applicable to specifications in all languages for which an appropriate transformation is available. On the other hand and more importantly, multi-language specifications are transformed into a homogeneous representation. This allows analysis across former language boundaries.

Language-specific transformation rules can be defined for specifications written in languages based on formal models of computation (MOCs) such as *synchronous data flow* [10] or *state charts* [5]. Due to the well-defined coordination semantics of formal MOCs, once defined these rules can be automatically applied to capture the relevant application properties. For specifications written in implementation languages, e. g. *C*, property extraction relies on designer knowledge but can be supported by code-analysis tools such as SYMTA [21]. For applications available as IP (intellectual property), the relevant application properties have to be part of the IP specification.

The second key element to reduce application complexity is to first analyze each process and channel separately. In other words, resource sharing effects are not considered at this point. For processes, conservative bounds on the core execution time (i. e. without interrupts) are needed. In case of a software implementation, static path-analysis combined with simulation of basic blocks and conservative combination of basic block timing results provides conservative bounds [13]. Tighter bounds can be obtained if simulation results are combined for segments with a single feasible path across basic block boundaries [21]. If a processor has a cache, worst- and best-case cache states at the boundaries of single-path segments also have to be considered.

For processes implemented in hardware, timing of sequential circuits is provided by synthesis tools, while more complex (combinatorial) circuits require designer knowledge, or in case of IP an appropriate description. For channels, the only information needed to be able to calculate communication delay is the amount of communicated data and the bus propagation delay.

Mapping-dependent properties for processes can be refined to include communication regions (intervals inside the core execution time interval, during which the process may be communicating), exclusive resource access regions or cache usage regions. This information can be provided if needed during system-level analysis (Section 3.4).

If distinct process behaviors are identified, then narrower intervals can be obtained for each of them using above techniques compared to a nondescript process abstraction that combines all possible behaviors. In Section 3.4 we will show how distinct process behaviors can be exploited using context information to obtain tighter bounds on system properties during system-level analysis.

3.3 Resource Interaction Abstraction

In the preceding section, we demonstrated how process interaction can be represented by only a few key parameters describing the externally observable behavior rather than detailed internal functionality. We did so by analyzing each process separately from all other processes. This was only possible because we ignored resource sharing effects.

Now, we will show how to abstract the interaction between the resources in order to analyze their performance, or more precisely, the influence of scheduling on the performance of each process. The same underlying ideas apply to communication resource analysis, too. However, the parameters are usually simpler than for process scheduling analysis. Therefore, we focus on process scheduling.

As mentioned in Section 2, there is no sufficiently general approach to analysis of complex SoC. However, there exists a host of work for dedicated sub-problems. Most of the work focuses on the analysis of processes which share a single resource. Therefore, we will introduce the procedure for single-resource process scheduling first.

Regardless of the actual scheduling strategy, all formal scheduling analysis techniques share some key properties. First, they are based on core execution times as the only process performance parameter; upper and lower bounds are provided by the individual process analysis introduced in the previous section. Secondly, the analysis techniques provide response times for the processes, i. e. the time between the activation of a process until the time of completion. And thirdly, they do so by formally capturing and evaluating information about the external signals which activate the execution of processes.

The external signals determine the system workload. Every scheduling analysis technique makes assumptions on the system environment which generates this workload. These assumptions are the event frequency, and how much data is provided per event. One can distinguish different types of events, such as the arrival of different packet types (control or data) in a packet network. The event frequency, the event types and the distribution in time are the main environment parameters.

Typically, the numerous input event models used in practice are classified into four classes (e. g. [17]).

- periodic: events arrive in equidistant periods of time (T)
- periodic with jitter: here, the events arrive generally periodic with the period T. However, each event may be delayed within a so called jitter window of size J.
- periodic with burst: here, n (burst size) events with a minimum distance of t (minimum inter-arrival time) arrive within a period T
- sporadic: in contrast to all other models, sporadic events – either burst or not– are not generally periodic. Therefore, only a minimum inter-arrival time t between two successive events is given.

Given these models for the input events or signals, scheduling analysis can be performed for one resource regardless of any other architecture component in the system. Thus, the event models abstract the resource interaction (step 3 in Fig. 2).

Now, we can analyze the scheduling influence for each resource individually. As a result, we obtain process (and communication) response times which are annotated back (step 4 in Fig. 2) to the system-level representation.

So far, we only investigated the abstraction of interaction between individual resources. As mentioned earlier, there is a small number of approaches to analyze more complex architectures, e. g. [16, 23]. It is advantageous to treat such multi-component subsystems as atomic with respect to performance analysis. Otherwise, the benefits of the approaches can not be exploited. Clearly, we have to identify such sub-systems within the overall system architecture first. Then, we can cluster such sub-architectures, and treat them in the same way as we treated single resources above. Again, the only information about the environment of the sub-system are event models.

3.4 Validation Flow

As already mentioned, process interaction abstraction and resource interaction abstraction are orthogonal concepts. This leads to the following basic validation flow (Fig. 2).

1. Process interaction abstraction is performed as described in Section 3.2 (step 1 in Fig. 2).
2. Process performance parameters are back-annotated to the system model (step 2 in Fig. 2).
3. Resource interaction abstraction is performed as described in Section 3.3 (step 3 in Fig. 2).
4. Resource performance parameters are back-annotated to the system model (step 4 in Fig. 2).

In Section 3.3 we showed that event models decouple the interaction of processes on different resources. Of course, as some point this interaction has to be accounted for.

A process on one resource generates output signals which are input to another resource. The output of signals or events can also be captured using event models. In [18], we have shown that these output event models are within the four input event models introduced above. Furthermore, we provided rules to derive output event models from the already known performance parameters (input event models and response times). We can easily couple the performance analyses for individual sub-systems by propagating event models through the architecture components along paths of process dependencies (step 5 in Fig. 2). An output event model of one component becomes an input event model for the connected component. This leads to an iterative procedure consisting of steps 3, 4, and 5.

While this is comparatively simple for feed-forward systems, it is more complex for systems with cyclic event model dependencies. In the presence of process dependency loops, we have no dedicated starting point with all event models given. Rather, we have to make assumptions about the actual parameters of certain input event models to start the iterative procedure. Furthermore, we have no dedicated termination point comparable to the system outputs in purely feed-forward systems. Thus, we have to iterate until all event models either converge or diverge. Convergence means that we have found a valid resource interaction abstraction. Divergence means that such an abstraction cannot be found.

Cyclic process dependencies are not the only source for cyclic event model dependencies. Due to resource sharing influences, also functionally independent processes may influence each others timing, and thus their corresponding output event models. In certain cases, the superposition of process dependencies and scheduling dependencies can result in additional cyclic dependencies of the corresponding event models. However, the impact on the overall analysis problem is the same as before.

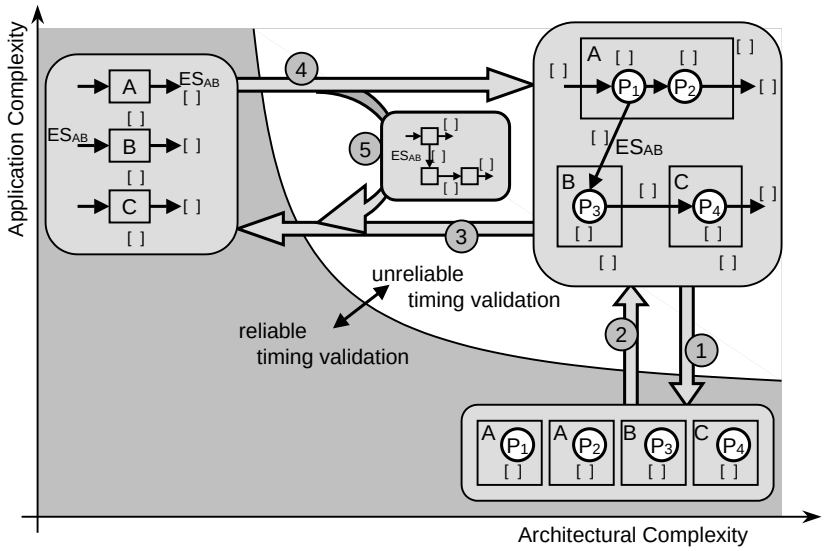

Figure 2: Proposed timing validation methodology

The basic steps 1 through 5 can be further refined for tighter bounds by using system-level information to exploit distinct process behaviors. A useful technique is to model *scenarios* (e. g. 'idle', 'connect', 'active' or 'disconnect' scenarios in a simple telephony protocol): only a subset of processes is active in each scenario, and only a subset of each process's behaviors is executed. Process behaviors have to be tagged with information about scenarios, in which they can be activated. This reduces the number of processes to consider and the size of intervals in each scenario during system-level analysis.

An additional technique to improve analysis results is to separate qualitatively different activation conditions of one process in the same scenario, e. g. periodic activation of one behavior and sporadic activation of a different behavior. Separate activation conditions often can be expressed using narrower intervals compared to a superposition of activation conditions.

Finally, in a practical design flow steps 3, 4 and 5 will be interleaved with steps 1 and 2 for efficiency reasons. Since mapping-dependent properties can be provided at various levels of detail, it is advantageous to initially provide only a basic view, in particular core execution times, and to defer obtaining more detailed information until explicitly asked for. For example, system-level analysis may ask at some point whether a process can request an exclusive resource within the 1*st* millisecond of its execution.

4 Design for Analyzability

So far, we introduced basic concepts and an appropriate validation flow to analyze performance of complex heterogeneous platforms. Event model propagation was identified as a key techniques in order to analyze the complex interactions between several resources. However, as the event model convergence problem in Section 3.4 already indicated, we have not yet discussed all problems related to event model propagation. In this section, we thus focus on the event model propagation step (step 5 in Fig. 2) in more detail.

So far, we implicitly assumed that the output event model of one component automatically matches the required input event model of the connected component. This is not necessarily the case. Recall that the scheduling strategies and corresponding analysis techniques determine which input event models are supported and which are not. Thus, even in the feed-forward case, event model propagation might fail at some point, since the output and input event models are incompatible. However, we can find simple event model interfaces (*EMIF*s) [17] to overcome such incompatibilities.

We introduce the basic idea of event model interfacing using a simple example. Consider the two processes $\mathsf{P_1}$ and $\mathsf{P_3}$ in Fig. 2, which exchange data via event stream $\mathsf{ES_{AB}}$. Other processes additionally to those shown in the figure can be mapped to both resources. On each resource, a different scheduling strategy is implemented. Let us furthermore assume, that the scheduling strategies for the two resources have the following properties:

- From the timing analysis of $\mathsf{P_1}$ we know that the output events are generally periodic with period T_1 but may experience a maximum jitter J_1 (e. g. resulting from preemptions by other processes).
- For the timing analysis of $\mathsf{P_3}$, the only available analysis approaches require a sporadic input event model with a minimum inter-arrival time t_3.

Clearly, the two event models can not be coupled directly since they are basically incompatible ($\mathsf{EM_{1,out}}$ =jitter→ $\mathsf{EM_{3,in}}$=sporadic). However, we are able to *derive the required* parameter values of the sporadic event model *from the known* values of the jitter event model: The minimum inter-arrival time of two successive events with jitter is the period minus the maximum jitter: $t_4 = T_1 - J_1$. Instead of propagating the event model (periodic with jitter) itself, we only capture the key characteristics of the event stream coming from $\mathsf{P_1}$ with respect to the parameters of the input event model of $\mathsf{P_3}$. Thus, $t_4 = t_3$. The mentioned parameter transformation represents the event model interface (*EMIF*).

Event model interfaces for other event model combinations can be found similarly [17]. Such *EMIF*s substantially enhance the event model propagation (or coupling) process, although the actual implementation does not change. Only the design representation, or more precisely, the event stream representation does. In other words, modifying the design representation can significantly increase the analyzability of the overall system. In the next paragraphs, we will see how slight modifications in the design (or implementation) itself can provide additional analysis capabilities, in cases where no simple *EMIF*s can be found.

Not all event model combinations are interfacable by such comparatively simple *EMIF*s as introduced above. As a counter example, consider the transformation ($\mathsf{EM_{1,out}}$ =jitter→ $\mathsf{EM_{3,in}}$ =periodic). It is trivial to prove that no general interface can be found: The event model $\mathsf{EM_{3,in}}$=periodic assumes the events to occur with equal temporal separation, which in general is not the case since $\mathsf{EM_{1,out}}$ contains a jitter. Only in the special case when $J_1 = 0$ an *EMIF* can be provided.

However, in digital signal processing systems, internal events are often assumed periodic since the system's overall input is purely periodic. But due to resource sharing and/or data-dependent process execution times, internal events experience a jitter. To keep

up with periodic models, buffers and timers are widely used to re-synchronize such events according to the initial period. This shows that it is generally possible to couple initially incompatible event models for analysis at the cost of additional system functions. We refer to these functions as event adaptation functions (*EAF*s).

The actual functionality of these *EAF*s can be derived from the two event models involved. For the above mentioned simple example ($\mathsf{EM}_{1,\mathsf{out}}$ =jitter$\rightarrow$ $\mathsf{EM}_{3,\mathsf{in}}$ =periodic), the *EAF* is straightforward. A buffer of size 1 and an output issue period of T_3 is needed. This way, we obtain a periodic output event model for the *EAF*. The actual period is simply determined by $T_3 = T_1$, which represents the *EMIF* to couple the two periodic streams.

In general, the parameters of the timed buffers (buffer size, issue rate, and the *buffering delay*) need to be formally derived from the model transformations. Since this paper focuses on methodology, we refer to [17] for the formal background. We introduced *EAF*s to couple initially incompatible event streams in cases, when simple *EMIF*s cannot be used. Clearly, this comes at the cost of additional system functions. However, most *EAF*s consist of a buffer and a timer only, usually not stealing too much system performance. Furthermore, the insertion of *EAF*s reflects what an experienced designer would do in manual design. In contrast, we showed how *EAF*s can be generated automatically with the same quality. This substantially improves design quality with respect to performance analysis (*"Design for Analyzability"*), and thus reduces the number of re-design cycles and overall design time.

5 Conclusion

In this paper, we have proposed a validation methodology which augments existing cosimulation-based design approaches with the capability to formally validate system-level timing. The basic idea of our methodology is to decompose the problem of system-level timing validation into the orthogonal concepts of single process and single resource analysis. This allows to use existing tools or approaches from industry and academia which are available for these steps. It has been shown how the results of the applied tools then can be combined in order to determine previously unobtainable system-level timing parameters such as end-to-end latencies. Furthermore, it has been shown how the analyzability of system timing can be systematically improved by slightly changing system implementation.

While the focus of this paper has been on the validation of system-level timing for complex SoC, the presented ideas are also valid for distributed systems such as networked electronic control units (ECU) in automotive applications. Other non-functional properties such as power consumption or buffer sizes can be validated analogously.

We have indicated the use of context information in order to model certain execution scenarios. A more comprehensive approach would be the extension of the event models in order to include context information. This would allow to analyze transitions between scenarios based on event models.

References

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

[2] CoWare. *CoWare N2C*. http://www.coware.com/cowareN2C.html.

[3] R. Ernst and A. A. Jerraya. Embedded system design with multiple languages. In *Proceedings Asia South Pacific Design Automation Conference (ASPDAC '00)*, pages 391–396, Yokohama, Japan, January 2000.

[4] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[5] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[6] C. Haubelt, J. Teich, and K. Richter. System design for flexibility. In *Proc. of Design, Automation and Test in Europe Conference (DATE'02)*, Paris, France, March 2002.

[7] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, pages 53–70, January 1999.

[8] 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.

[9] 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.

[10] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

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

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

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

[14] 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.

[15] T. Lundquist and P. Stenström. Integrating path and timing analysis using instruction level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[16] 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.

[17] 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.

[18] K. Richter, D. Ziegenbein, R. Ernst, L. Thiele, and J. Teich. Model composition for scheduling analysis in platform design. In *submitted to Proceeding 39th Design Automation Conference*, New Orleans, USA, June 2002.

[19] 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.

[20] 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.

[21] F. Wolf and R. Ernst. Intervals in software execution cost analysis. In *Proceedings 13th International Symposium on System Synthesis*, pages 130–135, Madrid, Spain, September 2000.

[22] F. Wolf and R. Ernst. Execution Cost Interval Refinement in Static Software Analysis. *The EUROMICRO Journal, Special Issue on Modern Methods and Tools in Digital System Design*, 47(3-4):339–356, April 2001.

[23] T. Yen and W. Wolf. Performance estimation for real-time distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11), November 1998.

[24] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI – A system model for heterogeneously specified embedded systems. *to appear in IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2002.