16 SPI - Workbench for the Analysis of Embedded Systems

M. Jersak*, K. Richter*, D. Ziegenbein*, R. Ernst*,
C. Haubelt**, F. Slomka** und J. Teich**

Zusammenfassung

Gemeinsamer Vortrag des Instituts für Datentechnik und Kommunikationsnetze (IDA), Prof. Rolf Ernst, TU Braunschweig, und des Instituts für Datentechnik (DATE), Prof. Jürgen Teich, Uni Paderborn. An den beiden Instituten entsteht in internationaler Kooperation mit Gruppen an der ETH Zürich, CH, (Prof. Thiele) und in Princeton, USA, (Prof. Wayne Wolf) die SPI-Workbench, deren aktuellen Stand vorstellt wird. Bei der SPI-Workbench handelt es sich um einen Werkzeugverbund für die Analyse eingebetteter Systeme. Die Besonderheit des SPI-Projektes liegt in der speziellen Betrachtung heterogener Systeme und heterogener Zielarchitekturen.

Hauptziel der Forschungsaktivität ist es, die systemweite Analyse für heterogene Systeme zu ermöglichen und somit die sichere Integration unterschiedlicher Systemteile und die sprach- und komponentenübergreifende Gesamtoptimierung unter Berücksichtigung nichtfunktionaler Systemanforderungen (z.B. Performanz, Flexibilität) zu erlauben. Die Grundlage für das Erreichen dieser Ziele stellt das gemeinsame, interne Systemmodell *SPI (System Property Intervals)* dar, das eine Abstraktion der verschiedenen Systemteile bezüglich ihrer Struktur und Eigenschaften darstellt und als Ausgangspunkt für die systemweite Analyse fungiert. Der Vortrag stellt einleitend die Grundidee der SPI-Workbench sowie das SPI-Modell vor. Danach wird im Detail auf die lokale und globale Zeitanalyse basierend auf dem SPI-Modell eingegangen, und schließlich beispielhaft die Modelltransformationen einer Eingabesprache in das SPI-Modell sowie die Exploration von Zielarchitekturen vorgestellt.

16.1 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-

* SPP 1040, Institut für Datentechnik und Kommunikationsnetze, TU Braunschweig, Hans-Sommer-Str. 66, D-38106 Braunschweig

** SPP 1040, Fachgebiet Datentechnik, Universität Paderborn, Warburger Straße 100, D-33098 Paderborn

specific languages with possibly fundamentally different underlying models of computation [1]. 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 (Figure 16-1).

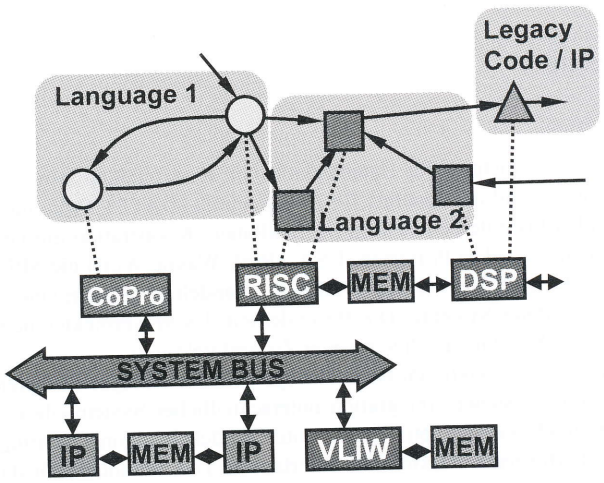

Figure 16-1: Typical embedded system architecture.

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 to determine 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 and test are the preferred means to system-level function validation. Here, test 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, simulation and prototyping were assumed to cover timing analysis as a side effect of system function validation. Specific timing analysis was at most used for safety critical or high availability systems or 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 have revealed

run-time anomalies [2] 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 [3]. 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 a burst of data in a certain situation, which is sent over a shared bus and delays transmission of other time-critical data. If the system integrator is aware of this particular problem, he/she might be able to add new simulation patterns. However, it is clear that for sufficiently complex systems, identification of all system-level corner cases is no practical option. Hence, VCC and related simulation tools are inappropriate for the upcoming problems of complex system performance analysis. Alternative approaches using statistical or typical performance data are increasingly unsafe due to 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.

However, system complexity and heterogeneity are also major obstacles for the system-wide application of formal validation methods. As presented in an overview of existing work in Section 16.2, reliable validation of system timing is usually only available for systems with either a low application complexity (e.g. single process) or a low architectural complexity (e.g. single resource). However, today's 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.

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

1. 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?
2. How can we implement systems in order to increase their analyzability?

Beyond the central question of system-level timing analysis, the paper introduces model transformations from standard system-level design languages into a representation suitable for analysis, and presents techniques for design-space exploration.

The paper is organized as follows. After an overview of existing approaches to timing analysis on various levels of abstraction in Section 16.2, the SPI workbench as our framework for analysis and exploration is presented in Section 16.3. The proposed validation methodology is introduced in Section 16.4. It builds upon a suitable system representation (Section 16.5) and consists of single-process analysis (Section 16.6), single-resource analysis (Section 16.7) and combination of results on the system-level (Section 16.8). Then, a systematic approach to adapt a system implementation for increased analyzability is presented in Section 16.9. Examples for model transformations from standard system-level design languages and design-space exploration are given in Sections 16.10 and 16.11, respectively. The paper is concluded by a summary and an outlook on future work.

16.2 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 [4] 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 [5] or require complex modifications to execution time determination [6]. Very few approaches like [7] 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 (*SYM*bolic *T*iming *A*nalysis) tool suite [8] 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 that influence input data dependent control structures. This way, subsets of 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, a large amount of work exists 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 [9]. They considered a static (Rate Monotonic) and a dynamic (Earliest Deadline First) priority assignment and provided a formal analysis framework for both. In [10], 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. [11], and arbitrary deadlines and burst [12] for static-priority scheduling. Sprunt et al. [13] analyze the influence of sporadic process activation.

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 [14] analyzing response times for static priority process scheduling combined with a TDMA bus protocol. There is no sufficiently 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 between 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 [23].

Such a compositional approach requires a suitable representation of the SoC's functionality, architecture and implementation choices (e.g. scheduling decisions), as well as a suitable representa-

tion of the environment of the system. We propose a novel system model called SPI (*S*ystem *P*roperty *I*ntervals) [17, 21], which enables global system analysis and system optimization across language boundaries, in order to allow reliable and optimized implementation of heterogeneously specified embedded real-time systems. The core SPI model mainly captures the application view of a heterogeneous SoC, and is extended by implementation and event models for efficient analysis.

16.3 The SPI Workbench

In this section, the concept and structure of the *SPI* (*S*ystem *P*roperty *I*ntervals) [17, 21] *workbench* are presented. Input is a system with its system function captured and optimized in application specific languages. The advantage of such a multi-language representation compared to using a uniform system specification language is the possibility to utilize specialized domain-specific design environments, optimization techniques and tools to increase productivity.

The SPI model (Section 16.5.1) serves as an internal abstract representation of the mixed system function specifically targeted to analysis. For this purpose, all relevant information is abstracted from the input languages and transformed into the semantics of the SPI model. A crucial point for the applicability of the SPI workbench is the availability of transformations from widely used standard languages to the SPI model. The principle of this translation and transformation has already been shown for different standard languages and models like SDF, periodic processes, VHDL processes, or StateCharts [15]. In section 16.10, a language transformation which has been developed and implemented for Matlab/Simulink [16] is presented.

The extraction of relevant aspects from the input descriptions and the mapping to the SPI model is dependent on user information regarding the level of abstraction in the extraction step and the granularity in the mapping step. Using the concept of process modes, the degree of abstraction used in parameter extraction can be controlled. The extreme cases are specifying a mode for each possible execution path of a process (high accuracy of modeling but exponential growth of paths with number of branches) or specifying a single behavior using uncertainty intervals (low accuracy but problem size reduction). Thus, during parameter extraction, a tradeoff between problem size and modeling accuracy which is directly related to the accuracy of the results is possible.

The SPI model is not a universal specification language but models a set of applications only in so far as it is relevant for analysis. In particular, information captured includes communication and activation conditions. Therefore, SPI can be viewed as a *coordination language* that captures the subsystems' internal process interaction as well as the coupling of the different subsystems. The functional description of the SPI processes may be given in completely different *host languages*.

For system-level analysis, the SPI representation is extended by appropriate architecture and environment models, and mapping and scheduling decisions are considered. It has been shown that standard scheduling techniques like periodic LCM-scheduling [17] can be applied to SPI graphs. A first approach to scheduling of an extension of the SPI model that combines static and dynamic scheduling can be found in [18]. A dynamic scheduling approach based on EDF (earliest deadline first) has been developed for SPI representations with fixed communication [19].

16.4 Methodology

In this section, our 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 [3] or CoWare [20]).

It has been shown that a major obstacle for 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 16.5.

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 [3]), or based on automated multi-objective optimization techniques (e.g. [21] and Section 16.11). 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 implementations will remain [21]. 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 (Section 16.6) and of the resource interaction (Section 16.7). The orthogonality of these concepts allows the composition of the obtained results (Sections 16.8 and 16.9).

16.5 Analysis Model

In this section, we informally introduce our 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 [8].

16.5.1 SPI Model

The SPI (*S*ystem *P*roperty *I*ntervals) model was developed to facilitate system-level analysis of multi-language designs, in particular system-level timing analysis [21]. In the SPI model, a system is represented as a set of concurrent processes which communicate tokens via unidirectional channels that are either FIFO queues (destructive read) or registers (destructive write). Processes as well as channels are not characterized by their exact internal functionality but by their abstract external behavior. This includes process activation conditions and the amount of communicated data per execution. Due to this simplicity, application specifications in different languages or models of computation can be captured with the same few SPI elements. Therefore, analysis of multi-language designs becomes feasible.

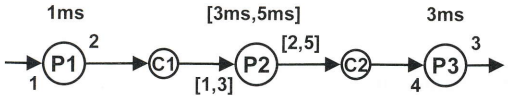

Figure 16-2: SPI example.

For example, process P_1 in Figure 16-2 consumes 1 data token and produces 2 data tokens per execution. By default P_1 is activated, i.e. ready for execution, if there are enough tokens available for one execution (in the example at least one) on its incoming channel. A core execution time of *1ms* is annotated for simplicity, but in fact results from a mapping decision.

A key concept in SPI is the possibility to specify all properties as intervals. For example, process P_2 consumes at least 1 and at most 3 tokens from channel C_1 and produces at least 2 and at most 5 tokens on channel C_2 per execution. P_2 is activated if at least 3 tokens are available at its input. The use of intervals allows to conservatively bound a set of possible behaviors, and to apply system-level analysis techniques which operate with conservative bounds and thus cover the whole design space. This avoids testing the huge number of situations that have to be considered in simulation-based validation approaches.

Additional concepts in SPI include process modes to model distinct process behaviors; mode tags to model process mode correlations; virtual processes and channels to model the system environment, timing constraints and otherwise hidden scheduling dependencies.

16.5.2 Architecture Model

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 either associated with processors or a bus. 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.

16.5.3 Event Model

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. Analogous to SPI, which abstracts process- and communication behavior, event models abstract a (potentially large) set of possible event sequences into a single representation. This allows analysis techniques to consider the full range of possible event timing.

The numerous input event models used in practice can be classified into four classes [23]:

1. periodic: events arrive in equidistant periods of time T
2. 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.
3. periodic with burst: here, n (burst size) events with a minimum distance of t (minimum inter-arrival time) arrive within a period T
4. sporadic: in contrast to all other models, sporadic events are not generally periodic. Therefore, only a minimum inter-arrival time t between two successive events is given.

16.5.4 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.

16.6 Single Process Timing-Analysis

A key element to reduce analysis complexity is to first analyze each process (and channel) separately. As a result, conservative bounds on the core execution time (i.e. without interrupts by other processes) and on the amount of communicated data are obtained. In case of a software implementation, static path-analysis combined with simulation of basic blocks and conservative combination of basic block timing results provides those conservative bounds [4]. Tighter bounds can be obtained if segments with a single feasible path across basic block boundaries are identified and then simulated or measured as a single entity, e.g. using the tool SYMTA [8]. 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.

If distinct process behaviors (modes) 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 the following experiment, execution time bounds of several benchmarks were analyzed using SYMTA and compared to the results of simulation. For this reason, programs were selected where best case and worst case input data could be determined by hand to deliver the real bounds during simulation. Results for the benchmarks are shown in Table 16-1. Timing of single-feasible path segments was measured on a commercial SPARClite evaluation kit by inserting trigger points at the beginning and end of segments. A trigger point is implemented by a store of its line number to a defined trigger address. It is detected by a logic state analyzer and saved with a timestamp [8].

Table 16-1: Comparison of SYMTA path-analysis with real best- and worst-case execution time bounds. Note that the error is always conservative (analyzed min cycles < measured min cycles; analyzed max cycles > measured max cycles).

Benchmark	min cycles			max cycles		
	measured	analyzed	error	measured	analyzed	error
3D-image	34908	33874	2.96 %	37848	38037	0.50 %
Diesel	62944	61445	2.38 %	62994	63333	0.54 %
Fft	1498817	1494650	0.28 %	1499176	1499290	0.01 %
Smooth	3635651	3570227	1.80 %	4846511	4881135	0.71 %
Blue	3564938	3345041	6.17 %	316865761	346541760	9.37 %
check-data	80	65	18.75 %	431	435	0.93 %
Wheatstone	2928459	2880230	1.65 %	3369459	3378098	0.26 %
Line	514	381	25.88 %	1619	2035	25.69 %

16.7 Single Resource Timing-Analysis

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 the influence of scheduling on the performance of each process. The same underlying ideas apply to communication resource analysis, too.

As explained in Section 16.2, there is no sufficiently general holistic approach to analysis of arbitrary complex SoCs. 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. Second, the analysis techniques provide response times for the processes, i.e. the time between the activation of a process until the time of completion. Third, they do so by formally capturing and evaluating information about the external signals which activate 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 (Section 16.5.3). The operating system overhead introduced through task management and context switching also has to be considered. Given the process core execution times, operating-system overhead and 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 process- and resource-interaction. Analysis determines process (and communication) response times which are annotated back to the system-level representation. A simple example is given in Figure 16-3.

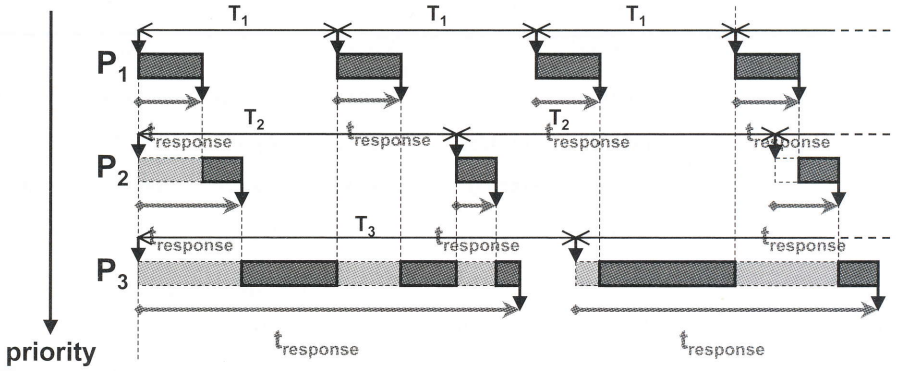

Figure 16-3: Rate-monotonic analysis example. For each process, its core execution time (here assumed constant for simplicity) and rate of periodic activation is given. The processes are dynamically scheduled by a real-time operating system using a preemptive, fixed-priority scheduler (scheduler overhead ignored for simplicity). Analysis then determines schedulability and the earliest and latest response time for each process.

There is a small number of approaches to analyze more complex architectures, e.g. [14, 24]. It is advantageous to treat such multi-component sub-systems as atomic with respect to performance analysis. Otherwise, the benefits of the approaches cannot be exploited. If we identify such sub-systems within the overall system architecture first, we can cluster and treat them in the same way as we treated single resources above.

16.8 System-Level Timing Analysis

We have explained in sections 16.6 and 16.7 that single process timing analysis and single resource timing analysis are orthogonal concepts. This leads to the following basic validation flow:

1. Single process analysis is performed as described in Section 16.6.
2. Process performance parameters are back-annotated to the system model.
3. Single resource analysis is performed as described in Section 16.7.
4. Resource performance parameters are back-annotated to the system model.

Event models decouple the interaction of processes on different resources (section 16.7). A process on one resource generates an output event model which is input to another process (potentially on a different resource). We can thus couple the performance analyses for individual sub-systems by propagating event models through the architecture components along paths of process dependencies. This leads to an iterative procedure consisting of steps 3 and 4, above.

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. Therefore, we have to iterate until all event models either converge or diverge. Convergence means that we have found a valid process- and 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 leads to additional cyclic dependencies of the corresponding event models.

16.9 Event Model Coupling

In this section, we present the actual coupling of event models. We first discuss compatibility issues between the four event models introduced in section 16.5.3, and derive event model interfaces (*EMIF*s) to transform the parameters of one event model into those of another model. In those cases, where no simple interface can be derived, we introduce event adaptation functions (*EAF*s) in order to couple initially incompatible event models. We introduce the basic idea of event model coupling using a simple example (Figure 16-4). Two processes P_1 and P_2 exchange data via event stream *ES*.

Figure 16-4: Event stream example.

Let us assume that the two processes are mapped to two different resources. Other processes are implemented on both resources, too, but they are not shown in the figure. Each resource implements a different resource sharing strategy. Let us furthermore assume, that the known analysis techniques for the two scheduling strategies have the following properties:

1. After timing analysis of P_1 we know that the output events are ‚periodic with jitter' with period T_1 and a maximum jitter J_1.
2. The timing analysis of P_2 that we want to apply requires a sporadic input event model with a minimum separation of t_2 between events.

The two local analysis approaches for P_1 and P_2 cannot be coupled directly since the two event models (jitter and sporadic) are incompatible. However, we can derive the required parameter values of the sporadic event model from the known values of the jitter event model: The minimum separation of two successive events with jitter is the period minus the maximum jitter: $t_2 = T_1 - J_1$. Event model interfaces for other event model combinations can be found similarly [23].

However, not all event model combinations are interfacable using *EMIF*s. As a counter example, consider the transformation $EM\{P_1, out\} = jitter \rightarrow EM\{P_2, in\} = periodic$. It is easy to see that in general no *EMIF* can be found: The event model $EM\{P_2, in\} = periodic$ assumes the events to

occur with equal separation, which in general is not the case since $EM\{P_1, out\}$ contains a jitter. Only in the special case when $J_1 = 0$ can an *EMIF* be provided. Instead, buffers and timers are needed to re-synchronize such jittery events into a periodic model. 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 the *EAF* is straightforward. A buffer of size 1 and an output issue period of $T_3 = T_1$ is needed. We refer to [23] for deriving the parameters of the timed buffers (buffer size, issue rate, and the buffering delay) for other event model transformations. Clearly, *EAF*s come 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. Here we show how *EAF*s with known properties can be generated automatically. This substantially improves design quality with respect to performance analysis (*„Design for Analyzability"*).

16.10 Language Transformation: Example Simulink

Simulink [25] is a block diagram oriented industry standard tool for simulating nonlinear dynamic systems that builds on the Matlab environment for technical computing. It uses a time-driven simulator that supports continuous-time, discrete-time (also multi-rate), or a hybrid of the two.

A Simulink block-diagram represents a set of differential equations. Directed edges between the blocks are used to communicate values. These edges have *register semantics* (non-destructive read, destructive write). Consequently, in multi-rate designs a value on an edge can be read multiple times (if the reading block is executed at a faster rate than the writing block), or it can be overwritten before having been read (if the reading block is executed at a slower rate than the writing block).

Simulink uses an idealized timing model for block execution and communication. Both happen *infinitely fast at exact points in simulated time*. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps. While this timing model is appropriate for numerical solutions of differential equations, it obviously cannot be implemented in an embedded system, since 1) it is fundamentally impossible to implement blocks or channels with zero execution time and 2) activation of all blocks at exact points in time is unnecessarily restrictive. Instead, there are usually only a few blocks where rates are critical, typically I/O blocks. Additionally, *latency path constraints* [26, 17] between inputs and outputs or along a cycle may have to be specified to guarantee timely completion.

During our translation to SPI, the restrictive time-driven activation of Simulink is thus replaced by data-driven activation. This relaxation preserves execution dependencies and allows to introduce those timing constraints that really matter, thereby increasing the design space for implementation. Details of this translation as well as a larger example can be found in [16].

16.11 Design-Space Exploration

We have explained that in embedded system design, not only functional requirements, but also a set of non-functional requirements such as timing, chip area, cost, power, etc. must be met. This added complexity calls for design-space exploration, where different possible solutions are selected for

analysis to obtain their performance tradeoffs. The goal is to find all pareto-optimal implementations. While exploration is already challenging for a single application, in areas such as platform-based design a system should be dimensioned such that it is able to implement not only one particular application optimally, but instead a complete set of different applications or variants of a certain application. Hence, we focus here on the task to find a tradeoff between the flexibility of the architecture that is able to implement several alternative behaviors and its cost.

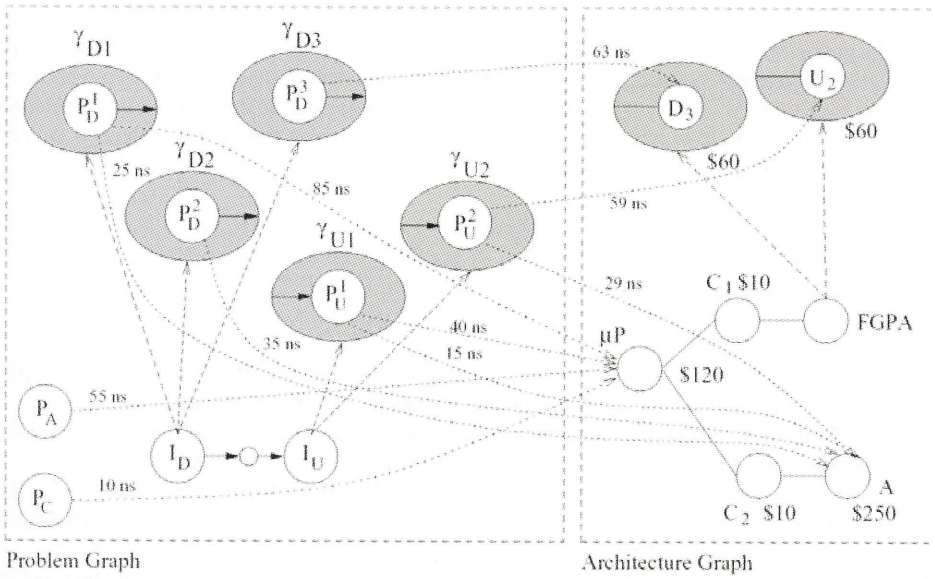

Figure 16-5: Hierarchical specification and analysis graph of a digital TV decoder.

As an example, consider the specification and analysis graph of a digital television decoder in Figure 16-5, which contains an application description using the SPI model (problem graph), an architecture graph including component cost, as well as possible mappings and the resulting core execution times provided by analysis. The main difference between different TV decoders is the implemented combinations of decryption and uncompression algorithms (hierarchical SPI elements I_D and I_U). We can try to support up to three decryption and two uncompression algorithms in this decoder (γ_{D1} - γ_{D3} and γ_{U1} , γ_{U2}). A possible architecture can be composed of a μ-Controller (μP), an ASIC (A), an FPGA, and two busses C_1 and C_2 to handle the communication.

During design-space exploration, different combinations of architecture components are allocated, and different mapping decisions taken. Each solution is evaluated with respect to architecture cost and flexibility to execute different alternative algorithms, each meeting its real-time requirements. Details of this approach as well as a larger example can be found in [21].

16.12 Conclusion and Outlook

In this paper, we presented the SPI workbench and a methodology for system-level timing validation. The basic idea of our methodology is to decompose the problem of system-level timing valida-

tion 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 the system implementation. Model transformations from standard system-level design languages into our representation as well as techniques for design-space exploration were introduced, which augment the core analysis functionality.

We are currently applying or planning to apply our system-level timing validation approach to several real-world systems, some specified using multiple standard design languages. Another focus is on decreasing over-conservative analysis by considering system contexts, such as correlations between process modes or execution scenarios. Design-space exploration will be extended to better consider different scheduling alternatives.

16.13 References

[1] Ernst, R., Jerraya, A. A. „Embedded system design with multiple languages“, in: Proc. Asia South Pacific Design Automation Conference (ASPDAC ’00), Yokohama (2000)

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

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

[4] Li, Y.-T. S., Malik, S. „Performance Analysis of Real-Time Embedded Software“, Kluwer Academic Publishers (1999)

[5] Healy, C., Arnold, R., Mueller, F., Whalley, D., Harmon, M. „Bounding pipeline and instruction cache performance“, IEEE Transactions on Computers, S. 53-70 (1999)

[6] Lundquist, T., Stenström, P. „Integrating path and timing analysis using instruction level simulation techniques“, in: Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES’98), Montreal (1998)

[7] Hergenhan, A., Rosenstiel, W. „Static timing analysis of embedded software on advanced processor architectures“, in: Proc. Design, Automation and Test in Europe (DATE’00), Paris (2000)

[8] Wolf, F., Ernst, R., Ye, W. „Path Clustering in Software Timing Analysis“, IEEE Transactions on VLSI Systems (2001)

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

[10] Kopetz, H., Gruensteidl, G. „TTP - a time-triggered protocol for fault-tolerant computing“, in: Proc. 23rd International Symposium on Fault-Tolerant Computing (1993)

[11] Sha, L., Rajkumar, R., Sathaye, S. S. „Generalized rate-monotonic scheduling theory: A framework for developing real-time systems“, Proc. of the IEEE, 82(1): 68-82 (1994)

[12] Lehoczky, J. „Fixed priority scheduling of periodic task sets with arbitrary deadlines“, in: Proc. Real-Time Systems Symposiom, S. 201-209 (1990)

[13] Sprunt, B., Sha, L., Lehoczky, J. „A periodic task scheduling for hard real-time systems“, Journal of Real-Time Systems, 1(1): 27-60 (1989)

[14] Pop, P., Eles, P., Peng, Z. „Bus access optimization for distributed embedded systems based on schedulability analysis“, in: Proc. Design Automation and Test in Europe (DATE 00), Paris (2000)

[15] Richter, K. „Developing a general model for scheduling of mixed transformative/reactive systems", Master's thesis, Institut für DV-Anlagen, TU Braunschweig (1998)

[16] Jersak, M., Cai, Y., Ziegenbein, D., Ernst, R. „A transformational approach to constraint relaxation of a time-driven simulation model", in: Proc. 13th International Symposium on System Synthesis, Madrid (2000)

[17] Ziegenbein, D., Ernst, R., Richter, K., Teich, J., Thiele, L. „Combining multiple models of computation for scheduling and allocation", in: Proc. International Workshop on Hardware/Software Co-Design (Codes/CASHE '98), Seattle (1998)

[18] Strehl, K., Thiele, L., Ziegenbein, D., Ernst, R. „Scheduling hardware/software systems using symbolic techniques", in: Proc. International Workshop on Hardware/Software Co-Design (Codes/CASHE '99), Rome (1999)

[19] Ziegenbein, D., Uerpmann, J., Ernst, R. „Dynamic response time optimization for SDF graphs", in: Proc. International Conference on Computer-Aided Design (ICCAD'00), San Jose (2000)

[20] CoWare, CoWare N2C, http://www.coware.com/cowareN2C.html

[21] Haubelt, C., Teich, J., Richter, K., Ernst, R. „System Design for Flexibility", in: Proc. of Design, Automation and Test in Europe Conference (DATE'02), Paris (2002)

[22] Ziegenbein, D., Richter, K., Ernst, R., Thiele, L., Teich, J. „SPI – A system model for heterogeneously specified embedded systems", IEEE Transactions on VLSI Systems (2002)

[23] Richter, K., Ernst, R. „Event model interfaces for heterogeneous system analysis", in: Proc. of Design, Automation and Test in Europe Conference (DATE'02), Paris (2002)

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

[25] The MathWorks, Simulink, http://www.mathworks.com

[26] Gupta, R. K., De Micheli, G. „Specification and Analysis of Timing Constraints for Embedded Systems", IEEE Transactions on CAD (1997)