SPI—A System Model for Heterogeneously Specified Embedded Systems

Dirk Ziegenbein, *Member, IEEE*, Kai Richter, *Member, IEEE*, Rolf Ernst, *Member, IEEE*, Lothar Thiele, *Member, IEEE*, and Jürgen Teich, *Member, IEEE*

***Abstract*—Embedded systems typically include reactive and transformative functions, often described in different languages and semantics which are well established in their respective application domains. Additionally, a large part of the system functionality and components is reused from previous designs including legacy code. There is little hope that a single language will replace this heterogeneous set of languages. A design process must be able to bridge the semantic differences for verification and synthesis and should account for limited knowledge of system properties. This paper presents the system property intervals (SPI) model, which employs behavioral intervals and process modes to allow the common representation of different languages and semantics. This model is the basis of a workbench which is targeted at the design of heterogeneously specified embedded systems.**

***Index Terms*—Embedded systems design automation, function variants, multilanguage design, process modes, property intervals, system modeling.**

I. Introduction

MANY embedded system applications consist of a combination of reactive and transformative functions. Often, several languages with different underlying models of computation are used in the design of an individual system. The languages are selected because of their particular suitability for certain applications and optimizations or because they have become generally accepted as a standard within an application field. The lack of coherency of the different languages, methods, and tools is a substantial obstacle on the way to higher design productivity and design quality. A similar problem occurs when reused components shall be integrated, possibly described in another language and incompletely documented. Examples would be intellectual property (IP) components "legacy code."

The key problems in the context of multilanguage design are the safe integration of the differently specified subsystems and the optimized implementation of the whole system. Both require the reliable validation of the system function as well as nonfunctional system properties. While there are several cosimulation-based approaches from both academia (e.g., [1]) and industry (e.g., [2]) that are well suited for functional validation and yield an easy understanding of the system function, these approaches fail to reliably validate nonfunctional constraints e.g., with respect to timing and power consumption. Since exhaustive simulation is infeasible in most cases, cosimulation can only cover part of the system behavior and thus, cannot guarantee that a corner case resulting e.g., in the worst case latency time of a process is covered.

We propose a novel system model called system property intervals (SPI) [3]–[5], 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. SPI is based on communicating processes whose behavior is described by a set of parameters. These parameters enable the adaption to different input models of computation. The consistency of these parameters is a prerequisite for global system optimization across the boundaries of different input languages.

A major step toward high-semantic flexibility of the model is the use of behavioral intervals, e.g., to describe input and output rates of processes. Sources of these uncertainty intervals may be data-dependent process behavior (e.g., due to `if-then-else` constructs in the input description), limited analyzability, or incomplete specification. Thus, behavioral intervals enable the integration of processes whose internal functional details are only partially known, i.e., "legacy code." On the other hand, using the concept of process modes, conditional process behavior depending on internal states or input data can also be modeled explicitly.

Due to the use of intervals for the communication behavior, a clear causal coupling of process executions may be lost resulting in limited optimization possibilities. Therefore, virtual processes and channels were introduced in the SPI model. These virtual components can be used to specify additional relations between processes concerning causality and timing constraints. In combination with behavioral intervals, these virtual components allow the modeling of the system environment as well as the representation and combination of a variety of models of computation in a consistent form. Since they are not part of the system functionality, they do not need to be implemented directlyl; rather, do they represent additional information for synthesis that e.g., might be implemented as part of a scheduler.

Many embedded systems are implemented with a set of alternative function variants to adapt the system to different applications or environments. The SPI model provides constructs

Manuscript received October 28, 2000; revised September 18, 2001. This work was supported in part by the German DFG Research Priority Program 1040.

D. Ziegenbein, K. Richter, and R. Ernst are with the Institute of Computer and Communication Network Engineering (IDA), Technical University of Braunschweig, Braunschweig, Germany (e-mail: ziegenbein@ida.ing.tu-bs.de; richter@ida.ing.tu-bs.de; ernst@ida.ing.tu-bs.de @ida.ing.tu-bs.de).

L. Thiele is with the Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland (e-mail: thiele@tik.ee.ethz.ch).

J. Teich is with the Computer Engineering Laboratory (DATE), University of Paderborn, Paderborn, Germany (e-mail: teich@date.uni-paderborn.de).

Digital Object Identifier 10.1109/TVLSI.2002.807767

to model the representation and selection of function variants of processes and process groups.

Based on the SPI model, an open workbench for the analysis and synthesis of heterogeneously specified embedded systems is currently being developed. A main incentive for the creation of such a workbench is the missing infrastructure in the field of system design automation. An open research platform with various opportunities for original contributions, exchange of algorithms and access to demonstrators seems to be a reasonable approach to this problem.

The remainder of the paper is structured as follows. After an overview of related work is given, the intended application of the model in the context of the SPI workbench is presented in Section III. In Section IV, the concepts of the SPI model are introduced and discussed, followed by an application example in Section V. The paper concludes with an outlook on future work.

II. Related Work

An outline of the state of the art in the area of system-level design is given in [6]. There, the insufficient coherence among the different languages, methods and tools is identified as a substantial obstacle on the way to a higher design productivity and to a reliable design process. An important aspect is the combination of domain-specific languages with their specific optimization procedures. Traditionally, flow-oriented models of computation are used in the area of signal processing or automatic control engineering. In the last two decades, models like synchronous data flow (SDF) graphs and Boolean data flow (BDF) have been established which are well suited for design space exploration, analysis, and synthesis. A very instructive comparison of such representations can be found in [7]. These models have already been accepted in industrial design driven by an extensive set of design tools which also support application optimization, i.e., COSSAP [8] or SPW [9]. A similar development can be observed for event-oriented models of computation, which are of special importance for reactive systems, e.g., STATEMATE [10] in automotive engineering or SDL [11] in the area of telecommunications. There are many other examples, mainly from academia, i.e., ESTEREL [12] and LOTOS [13], each of which represents a different model of computation.

In system-level design, programming languages like VHDL, C, C++, or Java are often used as a basis for the description of more abstract models. The commercial codesign system CoWare uses a client–server mechanism for process communication which is implemented in C-functions. C^x [14] and SpecC [15] use communicating processes with message passing via abstract channels whose behavior is described by C- functions. In the COSMOS codesign system [1], synchronous communicating finite state machines described in the SOLAR model are mapped to C or VHDL processes which, again, communicate via abstract channels. Communication between processes and channels is realized using an RPC mechanism. The RPC mechanism is also used in Matisse [16] where C++ methods enable process communication. CHINOOK implements message passing with Java methods [17] while in [18], C++ methods are used for the same purpose.

Languages like VHDL or SystemC [19] are obviously able to implement different models of computation enabling the common simulation of these models. For system synthesis, process communication has to be identified from such a detailed input description and abstracted, otherwise process scheduling as well as memory sizing are not possible. Exceptions are simple but less efficient scheduling techniques, which neglect communication like round robin scheduling or the traditional rate monotonic scheduling (RMS) [20]. Thus, for synthesis, common modeling at this low level of abstraction has no advantages.

Approaches to common modeling of substantially different semantics at a higher level of abstraction as needed for system synthesis are rare. The *charts [21] model combines finite state machines (FSM) and a variety of concurrency models (e.g., SDF) in an alternating hierarchy. Thus, FSM states can be refined by concurrency models and concurrency model nodes (e.g., data flow actors) can be refined by FSMs. Restrictions most notably concerning termination and communication of lower level models (e.g., an iteration defined by a minimum cycle for SDF graphs) allow propagation of properties like timing to higher levels of hierarchy resulting in a set of equations describing the system behavior. Similarly, the commercial CoCentric system studio [22] allows the combination of FSMs and data flow. However, instead of the well defined and restrictive integration semantics of *charts, CoCentric system studio allows not only hierarchical, but also parallel combination as well as a variety of termination semantics of nested models. This multitude of possibilities diminishes the analysis capabilities of the model in comparison to *charts.

Another approach based on the parallel composition of models of computation is the PCC model proposed in [23]. PCC differentiates event- and data-controlled processes with different activation mechanisms and introduces a partial order at the transitions between the different subgraphs in order to exclude possible nondeterministic behavior. This approach seems too restrictive for a common representation of a large number of different languages and models of computation.

In contrast to the above approaches, the proposed SPI model is not intended for unified functional specification but rather as a means to represent system properties relevant to analysis and synthesis in a homogeneous way. Thus, SPIs modeling concept naturally allows the representation of system parts without a systematic model of computation (e.g., legacy code) which seems to be a must for a complete design flow. However, SPI seems much less suited for functional verification which is a strength of *charts and PCC.

The FunState [24] internal design representation can be seen as a refinement of the SPI model that allows the explicit separation of data and control flow in subsystems called components. Thus, it seems to be a good choice for the representation of scheduling mechanisms and the intuitive visualization of state-based process behavior. A more comprehensive comparison of approaches to composition of different models of computation can be found in [25].

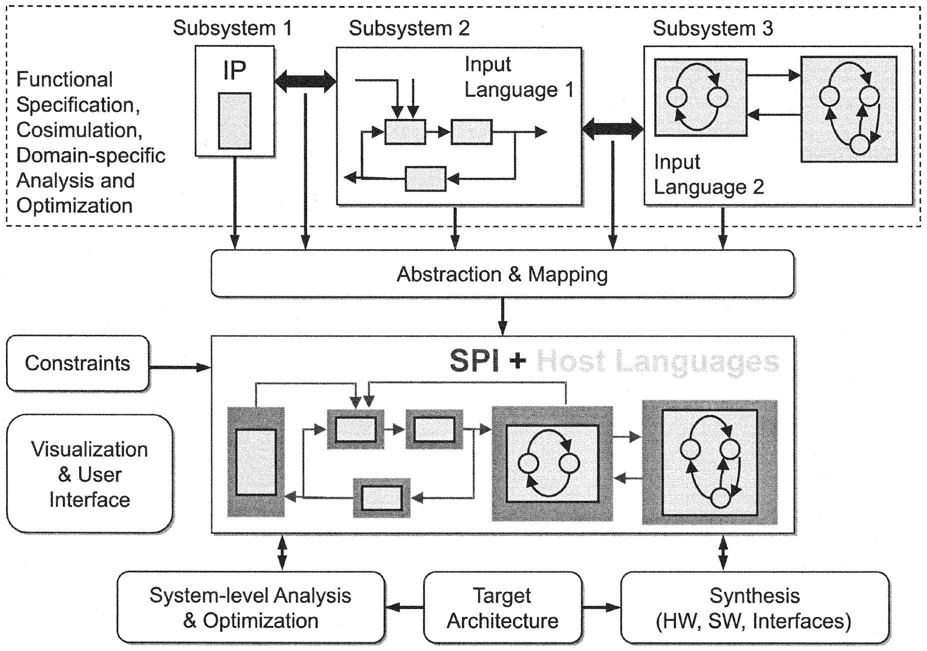

Subsystem 1
Subsystem 2
Subsystem 3
Functional Specification, Cosimulation, Domain-specific Analysis and Optimization
IP
Input Language 1
Input Language 2
Abstraction & Mapping
Constraints
SPI + Host Languages
Visualization & User Interface
System-level Analysis & Optimization
Target Architecture
Synthesis (HW, SW, Interfaces)

Fig. 1. Simplified structure of the SPI workbench.

III. The SPI Workbench

In this section, the concept and structure of the SPI workbench is presented. This workbench shall enable the analysis and synthesis of mixed reactive/transformative embedded systems described in several languages with possible differences in their underlying models of computation.

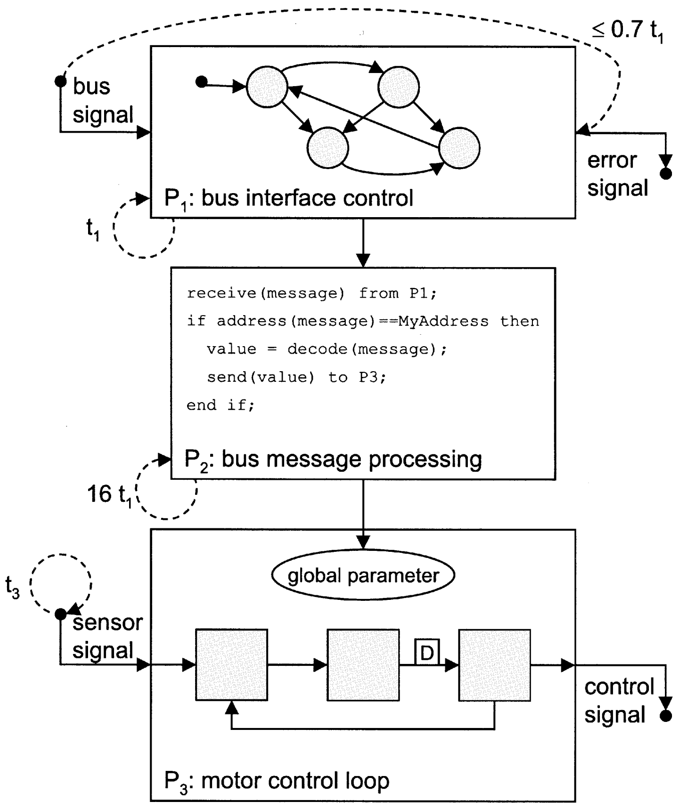
Fig. 1 shows the intended workbench structure that stresses the generation of a SPI representation and simplifies analysis and synthesis. Input to the SPI workbench is a system with its system function captured and optimized in application specific languages. The advantage of such a multilanguage representation compared to using a uniform system specification language is the possibility to include and utilize the traditional design environment of the application developer with domain-specific optimization techniques and tools.

The SPI model shall serve as an internal abstract representation of the mixed system function specifically targeted to synthesis. For this purpose, all information relevant to synthesis 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 Kahn process networks, SDF, periodic communicating processes, SDL, VHDL, or StateCharts [26]. Recently, a language transformation for Matlab/Simulink has been developed and implemented [27].

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 one 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 accuracy of modeling which is directly related to the accuracy of the results is possible.

The SPI model is not a universal specification language, but models the system behavior only in so far as it is relevant for synthesis, i.e., the resource utilization, the communication, and timing behavior. As can be seen from Fig. 1, 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 various host languages. This dualism between function and coordination is visualized by the light (host languages) and dark (SPI) gray shaded elements in Fig. 1.

Based on this SPI representation, system-level analysis and optimization regardless of language boundaries can be performed. It has been shown that standard scheduling techniques like periodic LCM-scheduling [3] can be applied to SPI graphs. A scheduling approach applicable to the SPI model that combines static and dynamic scheduling can be found in [28]. Furthermore, a dynamic scheduling approach based on earliest deadline first (EDF) has been developed for SPI representations with fixed communication [29]. The synthesis can be divided into the coordination synthesis (generation of interfaces, scheduler etc.) based on the SPI model and the functional synthesis (generation of functional blocks) based on the embedded host languages. Here, the goal is to reuse as much existing tools as possible, such as code generators for functional synthesis. The implementation of the Simulink transformation [27] that is based on the commercial real-time workshop code generator [30] shows that this goal is realistic.



```
receive(message) from P1;
if address(message)==MyAddress then
  value = decode(message);
  send(value) to P3;
end if;
```

Fig. 2. Remote motor controller.

This very comprehensive workbench is currently created and used by an international cooperation. Cooperation partners are research groups at the Technical University of Braunschweig, Germany, the Swiss Federal Institute of Technology Zurich, Switzerland, and the University of Paderborn, Germany. Also involved is a team at Princeton University, Princeton, New Jersey.

IV. The SPI Model

In this section, the concepts of the SPI model [3]–[5] are introduced. An example system will be used throughout the rest of the paper to demonstrate the features of the SPI model and the advantages of using a common internal representation.

This example system, is a remote motor controller which is specified as depicted in Fig. 2. The system collects message parts from a bus and tests them for an error (P_1), decodes the collected message (P_2), and sends a control word to the motor control loop (P_3). In this system, process P_1 is specified in a state-based language, process P_2 in a C derivative and process P_3, in a synchronous data flow language. The interaction of the processes and the environment is loosely defined. P_1 and P_2 are both periodic processes that are activated every t_1 and $16 \cdot t_1$ and have a deadline at the end of their period, while P_3 is driven by a periodic input. There is a timing constraint that constrains the maximum response time to an erroneously received message part to be no longer than 0.7 of the bus period t_1.

In the following sections, we will introduce constructs of the SPI model and often give examples each referring to parts of Fig. 6, which is a mapping of the remote motor controller to SPI.

A. Basic Model

The SPI model describes a system by means of processes communicating via unidirectional channels. There is no global communication, i.e., communication between processes has to be explicitly modeled in terms of channels. The channels are of two types, either first in first out (FIFO)-ordered queues (destructive read) or registers (destructive write). All channels have at most one writing process. But while registers may have more than one reading process (multireader registers), queues are constrained to have at most one reading process. This restriction prevents process activation conflicts due to the destructive read semantics and the data-driven activation principle (see Section IV-B).

This basic model can be represented by a *model graph* in which processes and channels are denoted as nodes connected by edges. Later on, this model will be extended by including additional parameters and incorporating hierarchical refinement (cluster graph).

Definition 1 (Model Graph): The *model graph* is a directed bipartite graph $G = (P, C, E)$ where

1) P denotes the set of *process nodes*;
2) $C = Q \cup R$ denotes the set of *channel nodes* with Q and R being the sets of queues and registers, respectively;
3) $E \subseteq (P \times C) \cup (C \times P)$ denotes the set of edges;
4) for each $q \in Q$, $indeg(q) = outdeg(q) \leq 1$, and for each $r \in R$, $indeg(r) \leq 1$. With $indeg$ and $outdeg$ being functions which return the number of a node's incoming and outgoing edges, respectively.

B. Execution Model

SPI is a nonexecutable model. Rather, a SPI representation bounds all possible behaviors of the system which it represents. In particular, we are interested in the activation, execution, and communication of processes.

The execution of a process is based on activation by data availability, i.e., a process is activated if its required input data is present and may execute if it is activated. Once a process is activated, its complete input data is present and the activation can not be invalidated by additional incoming data tokens.

Both computation and communication may consume time. The timing of a process may be described by a latency interval and communication regions for all edges connected to the process. In particular, the latency of a process denotes the time interval during which the required resource must be assigned to the process. A communication region denotes the time interval during which a process may send data on or receive data from the channel connected to the respective edge. Communication regions are specified relative to the start time of a process instance. These parameters obviously depend not only on the system specification but also on the resources the system is mapped onto and can be obtained by analysis methods like [31] (see Section IV-C).

The concept of communication regions enables a flexible adaptation of the process communication behavior in SPI, necessary to model several input languages and models of computation. A fixed communication scheme (e.g., read at start, write at end) is not flexible enough e.g., to capture C processes (legacy code) which can communicate at any time during their execution.

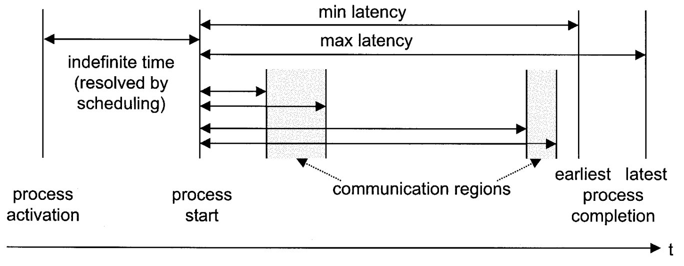

min latency
max latency
indefinite time (resolved by scheduling)
communication regions
process activation
process start
earliest process completion
latest process completion
t

Fig. 3. Qualitative timing of SPI process execution assuming nonpreemptive implementation.

The qualitative timing of a SPI process execution is shown in Fig. 3. The time between process activation and start of process execution is indefinite (due to the "*may start*"-semantics) and is resolved during scheduling. The diagram assumes nonpreemptive implementation i.e., a process once started is never interrupted e.g., by a higher priority process. However, this assumption is only made to clearly show the relation between process start, completion, and communication regions and is not a restriction of the SPI model.

C. Parameters

For scheduling, allocation, and performance analysis, knowledge about the detailed functionality of a process is not needed. It is sufficient to know for each process the properties related to resource requirements and interaction with its environment.

These properties of processes and channels are defined by parameters that are annotated to the corresponding graph elements. This allows an easy adaptation of the model to include all required information for a certain optimization goal or task in the design flow.

The parameters need not be fixed but can be specified using uncertainty intervals, i.e., they are constrained by an upper- and lower- bound. The sources for this nondeterminism can be abstraction of data-dependent functionality of a process (e.g., due to `if-then-else` structures) or limited analyzability of the input model on the one hand, or incomplete specification resulting in estimation of parameters on the other hand. While for the abstraction and limited analyzability, the parameter may switch between all possible values of the interval at runtime. The nondeterminism of the incomplete specification will be eliminated before runtime, such that it can be assumed that the parameter will take just one of the possible values of the uncertainty interval. For implementation-dependent parameters, limited analyzability of the target architecture (e.g., caches, out-of-order execution) is another source of nondeterminism. In the SPI model, however, the different types of nondeterminism are not distinguished since the differences that could be utilized by analysis and optimization methods could utilize are minimal.

In the following, we distinguish parameters based on their dependencies on application and target architecture. Functional parameters are solely dependent on the application while implementation-dependent parameters also depend on the target architecture.

Functional Parameters:

In order to determine communication and activation rates of processes, the communicated amount of data has to be known. Therefore, *data rates* denoting the number of data tokens communicated through a port at each process execution are specified.

Definition 2 (Data Rates): Let $Inputs(p) = \{c \in C | e = (c, p) \in E\}$ denote the set of *input channels* of process $p \in P$ and $Outputs(p) = \{c \in C | e = (p, c) \in E\}$ denote the set of *output channels* of p.

For each input channel $in \in Inputs(p)$, a process p has an *input data rate interval* $r_{in} = [r_{in,\min}, r_{in,\max}]$, which constrains the number of data tokens read from channel *in* per process execution. Analogously, for each output channel $out \in Outputs(p)$, there is an *output data rate interval* $s_{out} = [s_{out,\min}, s_{out,\max}]$. □

Note that the interval representation allows processes to have nonconstant data rates. Contrary to existing data flow models (e.g., synchronous data flow [32]) where data rates are fixed and time invariant, our representation allows for the modeling of nondeterminism to abstractly capture conditional behavior and incomplete specification.

Using process P_2 in Fig. 6, the meaning of data rate intervals can be easily explained. At each execution, P_2 consumes 16 tokens containing a message from channel R_2. Thus, r_{R_2} is equal to $[16, 16]$which is usually abbreviated as 16. If the message is for this controller, P_2 decodes the message and produces one token containing a control word on channel R_4. Otherwise, no control word and, thus, no token is written to R_4. Therefore, P_2's data rate interval for the production on channel R_4 is $s_{R_4} = [0, 1]$.

Furthermore, we need to define the initialization of channel contents as known from models of computations like dataflow process networks [7].

Definition 3 (Channel Initialization): Associated with each channel $c \in C$, there is the parameter $d_c = [d_{c,\min}, d_{c,\max}]$ denoting the initial number of data tokens on channel c. □

For the calculation of absolute values for the amount of communicated data or memory size requirements, the size of data tokens has to be specified.

Definition 4 (Token Size): Associated with each channel $c \in C$, there is a *token size* $b_c = [b_{c,\min}, b_{c,\max}]$ denoting the size of each token communicated on channel c. □

For modeling purposes, we introduce the concept of *virtuality* for processes and channels.

Definition 5 (Virtuality): Associated with each process $p \in P$ and each channel $c \in C$, there is a *virtuality flag* $v \in \{$"*true*", "*false*" which denotes the fact whether the process or channel is part of the system to be implemented ($v :=$ "*false*") or has been introduced for modeling purposes only ($v :=$ "*true*"). □

Virtual model elements are used to describe the system environment and to visualize implementation-relevant information like rate constraints (see Section IV-E) or relative execution rates [27]. Although, there is usually no direct implementation equivalent to a virtual model element, the visualized information imposes constraints on the design space (e.g., precedence constraints between processes) and has to be regarded during implementation (e.g., during scheduling). Virtual model elements are denoted by dashed lines in the graphical representation.

Implementation-Dependent Parameters:

Besides the implementation-independent behavior captured by the above constructs, properties like execution time, communication time, or power consumption are of central importance for system cost optimization. These properties do not only depend on the functional specification but also on the available target architecture and its resources. Thus, they are usually obtained by estimation or analysis [31].

Then, parameters describing these properties may be annotated to the SPI model elements. However, since the SPI model is to be independent of the target architecture, these parameters are not part of the core SPI model. Rather, the placement of this information should represent its dependencies not only on the functional specification but also on the target architecture. This is regarded in the current development of a specification graph that augments the SPI model with a model of the HW/SW target architecture and edges between elements of both models mapping functional elements to architectural resources. This approach is based on the model proposed in [33].

In the context of this paper, we show the intended use of implementation-dependent parameters by annotating latency time intervals lat to processes. Latency-time intervals denote the duration of a process execution on a certain resource and are used by the static scheduling approach applied to the example system in Section V. Evidently, communication regions are equally implementation dependent as the process latency. For reasons of simplicity, we assume in the context of this paper that processes always read at start and write at completion, although communication regions allow a more flexible communication behavior.

Note that implementation-dependent parameters of virtual-model elements are not defined since they do not have a direct implementation equivalent.

D. Process Modes

Fig. 4 shows a simplified-process network implementing the wireless IP standard on a pico cellular base station [34]. The solid arrows visualize different execution paths through the network depending on the context of the arriving data packets (e.g., packet from wireless to Ethernet). Depending on this context, processes show different behaviors e.g., process *if_output* sends a packet either to process *ether_output* or *radio_output* depending on the target information in the packet header.

With the SPI model as presented so far, a SPI process models the nondeterminism caused by different execution paths in the original process by means of uncertainty intervals. For process *if_output*, this means that a corresponding SPI process would have an input data rate of 1, while both output data rates would be [0, 1]. This interval abstraction may lead to too imprecise modelings, having not enough detail for efficient scheduling or successful validation. For the example, the fact that *if_output* will produce data only on one of its output channels and, thus, the correlation between both output data rates (*either* one token to *ether_output or* one token to *radio_output*) is lost. Therefore, the concept of process modes is introduced to be able to explicitly model different process behaviors and represent the correlation between process parameters.

Definition 6 (Process Modes): A *process mode* m_p of a process $p \in P$ is a tuple of a latency time interval[1] $lat_p = [lat_{p,\min}, lat_{p,\max}]$, an input data rate interval $r_{in} = [r_{in,\min}, r_{in,\max}]$ for each of its input channels $in \in Inputs(p)$ and an output data rate interval

[1]Only included in the context of this paper to demonstrate the mode-dependency of implementation-dependent parameters.

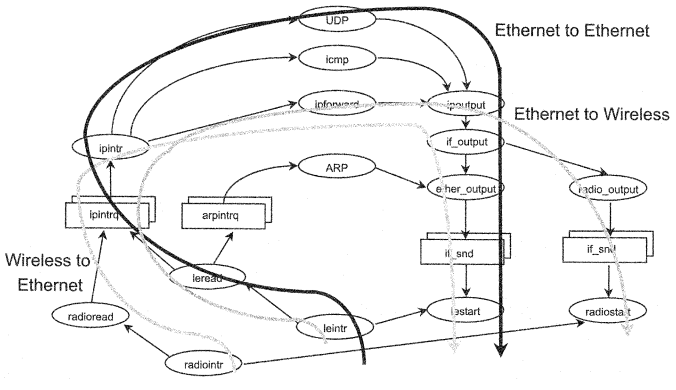


Fig. 4. Context-dependent flow of execution in a base station.

$s_{out} = [s_{out,\min}, s_{out,\max}]$ for each of its output channels $out \in Outputs(p)$.

Associated with each process $p \in P$, there is a nonempty, finite *mode set* $M_p = \{m_{p,1}, \ldots, m_{p,n_p}\}$ where $n_p \in \mathbb{N}$ is the total number of modes of process p. □

A process mode describes a subset of the possible process behaviors, i.e., a mode can be obtained for a subset of execution paths just like the single process behavior is obtained for the set of all possible execution paths of a process.

An example for a process with different possible execution paths is the bus interface control process of the example system in Fig. 2. Assuming this process is to be modeled using two modes, the different execution paths can be grouped as follows. The process either reads an erroneous message part from the bus and outputs an error message (mode m_1), or it reads a correctly received message part (mode m_2). Then, the process modes describing the corresponding SPI process P_1 in Fig. 6 are

$$\begin{aligned} m_i &= (lat, r_{R_1}, r_{Q_7}, s_{R_2}, s_{Q_3}, s_{Q_8}) \\ m_1 &= ([0.3, 0.6], 1, 1, 0, 1, 1) \\ m_2 &= ([0.4, 0.8], 1, 1, [0, 16], 0, 1). \end{aligned}$$

For simplicity, process latencies are expressed as fractions or multiples of t_1, the period of the incoming bus signals. Note that mode m_2 still contains uncertainty with respect to the communication behavior of process P_1 as denoted by the output data rate interval for register R_2. This interval represents the fact that this output is only produced if the received message is the last part of a telegram, i.e., the whole telegram is written to the decoder.

Usually (and in both examples above) a process adapts its behavior, i.e., selects its mode based on the content of its input data (wireless IP: target information in packet header, motor controller: erroneous or correct message part). So far, data has been abstracted to tokens that carry no information on data content or value. Thus, a set of *mode tags* modeling data content information may be associated with the data tokens.

Definition 7 (Mode Tags): A mode tag $(name, val)$ is a pair of a unique identifier $name$ and a value $val \in D(name)$ with $D(name)$ being the finite domain of $name$. □

Then, e.g., $(answer, 42)$ denotes that the tag $answer$ has the value 42 which abstracts information of the token the tag is associated to. Although, a tag may have only a single value at a time, it is possible to specify a set of possible values for a tag denoting uncertainty of the correct value at a certain time instance.

This corresponds to the use of intervals for parameters such as data rates. Please note that mode tags are a virtual concept i.e., they do not have to be implemented since they only visualize content of data that is already communicated.

In the following, constructs for the generation and evaluation of mode tags will be introduced. Since the specification of content information is only meaningful in correlation to the data containing the content, mode tag production has to be directly coupled with the production of data tokens. Thus, the production of mode tags modeled by *mode tag production rules* is associated with the processes producing the corresponding data. Furthermore, the definition of the channel initialization has to be extended to include a possibility to associate mode tags with the initial tokens.

Definition 8 (Production of Mode Tags): Associated with each process $p \in P$, there is a finite set of *mode tag production rules* TP_p. Each tag production rule is a mapping $tp_p\colon I_p \mapsto O_p$ where I_p is a set of input predicates and O_p is a set of output tag production patterns.

An input predicate is a function on the numbers of available tokens $in.num$ on input channels $in \in Inputs(p)$ and on the tag sets $in.tag(k)$ of the tokens read by p in its current execution where $in.tag(k)$ with $k \in \mathbb{N}^+$ denotes the tag set of the kth token in channel in. The value of the predicate is either "true" or "false." Each input predicate $i \in I_p$ maps to an output tag production pattern $o \in O_p$ that is activated if the value of i is "true." An activated tag production pattern associates mode tags with certain tokens produced on the output channels of the process. The tag set $out.tag(k)$ of the kth token produced on channel out during the current process execution is the union of all mode tags associated with this token by activated tag production patterns. □

Note, that the production of mode tags like the production of data tokens may be mode dependent. Thus, the definition of process modes m_p has to be extended to include mode tag production rules.

The following example demonstrates the use of tag production rules. Process P_{bus} in Fig. 6 models the production of bus telegrams that may be correct or faulty. This fact is denoted by a mode tag named $error$ that has the possible values "$true$" or "$false$" and that is attached to all produced tokens. Then, process P_{bus} has the following two modes representing the cases of producing a correct (m_1) or a faulty telegram (m_2)

$$\begin{aligned} m_i &= (lat,\, r_{Q_{12}},\, s_{R_1},\, s_{Q_{13}},\, TP)\\ m_1 &= (0,\, 1,\, 1,\, 1,\, \{\text{“}true\text{”}\\ &\qquad \mapsto def\,(R_1.tag(1),\, error,\, \text{“}false\text{”})\})\\ m_2 &= (0,\, 1,\, 1,\, 1,\, \{\text{“}true\text{”}\\ &\qquad \mapsto def\,(R_1.tag(1),\, error,\, \text{“}true\text{”})\}). \end{aligned}$$

Since the mode tag production depends only on the mode (m_1 or m_2) the predicates of the rules in both modes are simply "$true$." In general, the output tag production is described by a function $def(c.tag(k),\, name,\, value)$[2] that causes a mode tag $(name,\, value)$ to be added to the tag set of the kth token produced on channel c. Therefore, if P_{bus} executes e.g., in mode m_1 one token with the tag $(error,\, \text{“}false\text{”})$ is produced on R_1. Nothing can be said about the selection of the mode since there is no incoming data for P_{bus} besides tokens from the virtual queue Q_{12} only constraining the relative timing of P_{bus}.

[2]It is possible to specify a set of possible values for the $value$ parameter denoting uncertainty about the value of the produced tag.

How mode tags are utilized for process mode selection is part of the activation function that is described in the following section.

E. Activation Function

An *activation function* determines the conditions under which a process is activated and at the same time, determines the mode in which a process will execute.

Definition 9 (Activation Function): Associated with each process $p \in P$, there is an *activation function* that may be formulated as a finite set of rules σ where each rule is a mapping of an input predicate to a finite set of modes $M_\sigma \subseteq M_p$.

The input predicate of a rule σ is a function on the numbers of available tokens $in.num$ and on the tag sets $in.tag(k)$ of the tokens that will be read by p in the execution to be activated on input channels $in \in Inputs(p)$. The value of the predicate is either "true" or "false." The union of all sets M_σ with the predicate of rule σ being "true" is the set of modes the process can execute in. If and only if this set is not the empty set, the process is activated. □

If the activation function results in a set of several possible modes for an execution, then it is uncertain in which of the activated modes the process will execute.

The activation rules in its general form allow the specification of activation conditions that violate basic principles of the SPI model. Thus, in the following some restrictions for activation rules are defined. These restrictions do not impose additional constraints on the valid process behavior but rather allow a check for correct modeling.

Note that the predicate of a rule σ has to be "false" if in some input queue c there is a smaller number of available tokens than possibly consumed by the process in any mode $m \in M_\sigma$. Otherwise, the execution of this process may lead to an attempted consumption of nonavailable data and would violate SPI's data-driven activation principle.

Since SPI processes are not allowed to test for input data without reading it, process behavior, and thus process modes, may only depend on read input data. This is accounted for by the restriction for input predicates to only consider mode tags of tokens (to be) read during the current process execution.

A consequence of the data-driven activation and the forbidden test for input data is that an activation of a process can never be invalidated.

In examples, the activation function is often omitted for simplicity. In these cases, the default activation is that each mode is enabled (i.e., the process may execute in it) if there are enough input tokens available for an execution.

Evidently, the specified activation function is capable of representing data flow or event driven models of computation. In the following, it will be shown how time driven activation can be modeled using the SPI activation function.

Periodic activation (e.g., of process P_{time} in Fig. 6 neglecting the input channels Q_8, Q_{10}, and Q_{13}) can be modeled by a virtual channel Q_{11} starting and ending at the process to be activated. The process has a static consumption and production rate of one data token per execution for channel Q_{11} and, thus, an activation function consisting of a single rule $Q_{11}.num \geq 1 \mapsto$

m_1 mapping to the only mode of $P_{\rm time}$. With one initial data token on the channel ($d_{Q_{11}} = 1$) supporting the first activation, each execution now enables its following activation. The time between two consecutive executions can be constrained by a latency constraint on channel Q_{11} (to be introduced in Section IV-G).

Another possible application is the *activation by relative execution rates* (e.g., no exact periodicity but constrained mobility intervals as in the process model used in RMS [20]). An example for this is process P_1 in Fig. 6, which has to be executed once during every period t_1 and thus once between two executions of the virtual process P_{time} having an exact period of t_1. To model this fact, two virtual channels Q_7 and Q_8 with Q_8 having one preassigned token are introduced. With the modes of P_1 (as defined in Section IV-D), the activation function of P_1 consists of the following two rules [3]

$$(Q_7.num \geq 1) \wedge (is\,(R_1.tag(1),\, error,\, \text{“}true\text{”})) \mapsto m_1$$
$$(Q_7.num \geq 1) \wedge (is\,(R_1.tag(1),\, error,\, \text{“}false\text{”})) \mapsto m_2.$$

Then, the first execution of P_{time} causes an activation of P_1 whose execution in turn leads to another activation of P_{time} etc. The chosen mode depends on the value of the tag $error$ denoting a transmission error on channel R_1. In general, the function $is(ts,\, name,\, val)$ returns "$true$" if the tag set ts contains the information that tag $name$ has the value val.

F. Function Variants

Many embedded systems are implemented with a fixed-core function and a set of alternative *function variants* to adapt the system to different applications or environments. Examples are TV sets, which can be adapted to different standards or automotive control systems used in countries with different emission laws. Function variants are mutually exclusive, i.e., only one variant of a set of alternative functions is selected at a time. There may be several of those variant sets in one embedded system, e.g., for different input and output standards of a multimedia device. The variant selection for these sets may be related or independent.

Function variant selection can occur at different stages of a product life time corresponding to different variant types. *Production variants* are selected at production time, e.g., by downloading a certain software variant into an EPROM. *Run-time variants* are selected at system start-up time, e.g., as part of a boot sequence, which reads switches or flash memory stored parameters. In both cases, system optimization can assume that the system's variants can not be changed during system operation. A more complicated selection process is found in dynamically *reconfigurable architectures*. Here, a subsystem is typically configured by a higher level controller to execute a function which the subsystem itself cannot change.

Clearly, a single process with a set of modes can be used to represent function variants, where each of the variants is mapped to one or more modes of that process. Then, the variant selection maps to mode selection inside the process. The drawback of this representation is that the modeling is too coarse grain, since function variants usually incorporate several processes and channels, i.e., whole subgraphs, instead of a single process. To keep the level of granularity, the subgraphs of all function variants can be included in the system model together with some coordination framework that is responsible for the distribution of the data according to the currently selected subgraph (by means of mode tags). In this case, the information about the mutual exclusion of the function variants is distributed within the process network and can hardly be recovered. Thus, the use of processes and process modes is not sufficient for a reasonable representation of function variants.

[3]$(R_1.num \geq 1)$ does not have to be specified since registers always are "full" and thus a register access can not result in an attempt to consume unavailable data.

In the remainder of this section, we will only focus on *function variant representation*. For detailed information about reconfiguration and dynamic variant selection, we refer to [35], [5].

SPI Representation of Function Variants:

As already indicated, changing a system's variant in the functional description corresponds to exchanging subgraphs in SPI. Such subgraphs may be represented as *clusters*. A cluster contains a set of graph elements which communicate through the cluster border via input and output ports. This concept allows for hierarchical construction of complex SPI models and enables stepwise refinement.

Definition 10 (Cluster): A *cluster* is a tuple $\gamma = (I,\, O,\, P,\, C,\, E,\, \Psi)$ where I denotes the set of input ports and O the set of output ports, respectively. P denotes the set of embedded processes, C the embedded channels, and $E \subseteq ((P \cup \Psi) \times (C \cup O)) \cup ((C \cup I) \times (P \cup \Psi))$ the embedded edges. Ψ denotes the set of embedded interfaces to be defined later. In addition to the constraints on channels as given in Def. 1, an input port $in \in I$ satisfies $indeg(in) = 0$, $outdeg(in) \leq 1$ and an output port $out \in O$ satisfies $outdeg(out) = 0$, $indeg(out) \leq 1$. □

Clustering does not add functionality to the model and is only a structuring concept. The only restriction in this context is that a cluster, like a process, can only be connected to channels.

Now, a system with two function variants can be represented consisting of three parts. The first common part contains all elements that are not variant-dependent, while the remaining parts are mutually exclusive clusters which represent the distinct function variants. Evidently, both clusters must have the same external connections in terms of input and output ports, since otherwise they could not be reasonably exchanged by each other. In other words, the three parts need a common *interface*. Furthermore, information is needed about a reasonable *mapping* between interface ports and cluster ports.

Definition 11 (Interface): An *interface* is a tuple $\psi = (I, O, \Gamma,\, PortMap)$ where I denotes the set of input ports, O the set of output ports, Γ the set of clusters associated with this interface, and a partial function $PortMap$, which maps the input and output ports of each cluster in Γ to an input or output port of the interface ψ, respectively. If an interface is embedded into a cluster, the indegree of input ports and the outdegree of output ports is at most one. □

Using these two constructs, a system part for which different function variants exist can be represented by an interface with a set of different clusters associated. Then, each function variant is represented by exactly one of the clusters that can be connected to the interface according to the corresponding mapping in $PortMap$. An example for a system part with two-function

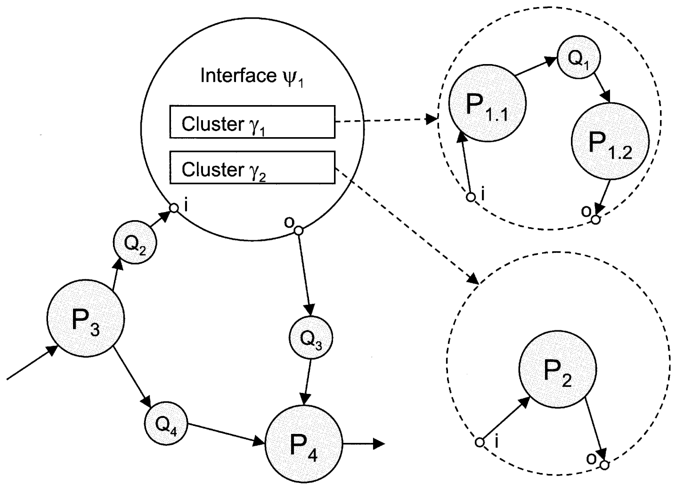


Fig. 5. Example system with two-function variants.

variants is the interface ψ_1 in Fig. 5. The different variants are depicted as clusters γ_1 and γ_2. The port mappings are not shown.

G. Environment Modeling

Evidently, the modeling of an embedded system has to include the system's environment and its constraints imposed on the system. Using virtual model elements, system and environment can be modeled in a coherent way. Examples for virtual processes modeling the environment are the source processes P_{bus} and P_{sensor} as well as the sink processes P_{error} and P_{motor} in Fig. 6. Note that the channels between the environment and the system are not virtual, since in this example this communication has to be implemented (e.g., using memory mapped I/O).

Another important part of the environment are *timing constraints*. In the SPI model, latency constraints can be specified for paths of channels and restrict the time tokens may take to travel along these paths.

Definition 12 (Latency Path Constraints):

A path constraint is a labeled path in the SPI graph. A path is of the form

$$(P_1 \longrightarrow C_1 \longrightarrow P_2 \cdots \longrightarrow C_n \longrightarrow P_{n+1})$$

while involving $n+1$ processes, n channel nodes and $2n$ edges. A *latency path constraint*

$$LC_{\text{path}} = [t_{\text{lat, min}}, t_{\text{lat, max}}]$$

denotes the time interval between the time a token is written to the first channel of the path C_1 and the time a causally dependent token is removed from the last channel of the path C_n. The path constraint must be satisfied for any token sequence during any possible execution of the system. □

A token v is said to be causally dependent on another token t, if it was produced by a process execution which either read t or a token which is causally dependent on t. For a constructive method to check these latency path constraints see [4]. Using virtual model elements, other types of timing constraints like periodicity constraints can be modeled using latency constraints (e.g., process P_{time} in Fig. 6).

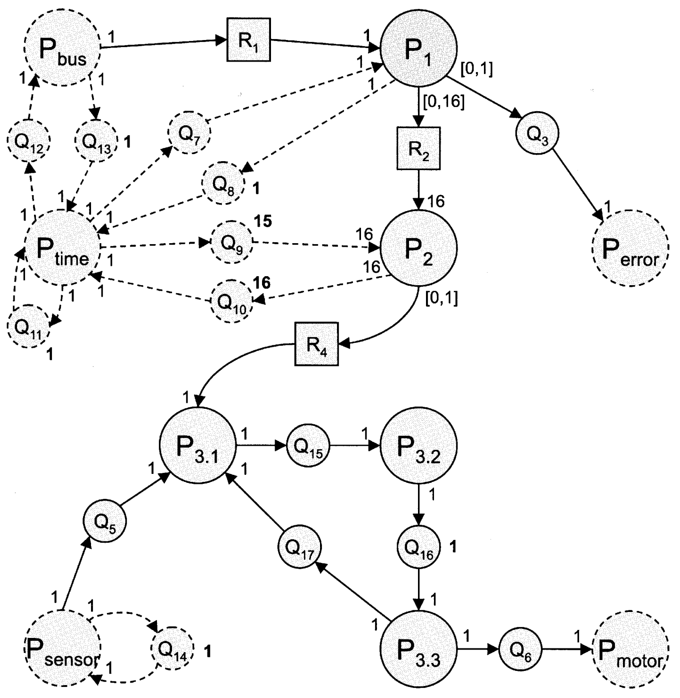


Fig. 6. Remote motor controller (SPI Mapping).

V. Application Example

In this section, an offline nonpreemptive scheduling method (e.g., [36]) is applied to the remote motor controller of Fig. 2. The corresponding SPI representation of this system is depicted in Fig. 6. Different parts of this representation have already been discussed throughout the paper while serving as examples for the different SPI model elements and concepts. Note that the SDF graph in process P_3 is mapped to three processes ($P_{3.1}$, $P_{3.2}$, and $P_{3.3}$) while the state machine in P_1 is mapped to a single process. These mapping decisions are left to the designer as part of a tradeoff between problem size and modeling accuracy. A suitable notation to provide this granularity information is currently being developed using the Matlab/Simulink translation as a demonstrator [27].

Another mapping decision is the representation of process P_1 with two modes as previously described in Section IV-D, instead of representing it using a single behavior. This behavior could be generated by merging the intervals of both modes and would be

$$\begin{aligned} m &= (\text{lat}, r_{R_1}, r_{Q_7}, s_{R_2}, s_{Q_3}, s_{Q_8}) \\ m &= ([0.3, 0.8], 1, 1, [0, 16], [0, 1], 1). \end{aligned}$$

When looking at the constraint limiting the response time to a transmission error that is modeled by the latency path constraint $LC_{\text{err}} = LC_{(P_{\text{bus}} \rightarrow R_1 \rightarrow P_1 \rightarrow Q_3 \rightarrow P_{\text{error}})} = [0, 0.7t_1]$, it becomes evident that the specification of P_1 using two modes is necessary in order to be able to guarantee the satisfaction of LC_{err}. With a single behavior we have to assume the maximum execution time 0.8 t_1 for P_1 which violates the maximum response time 0.7 t_1 specified by LC_{err}. However, when using the two-mode representation, we just have to consider the maximum latency of mode m_1 which is the mode that outputs the error message that is $0.6t_1 < 0.7t_1$ such that the satisfaction of LC_{err} becomes possible.

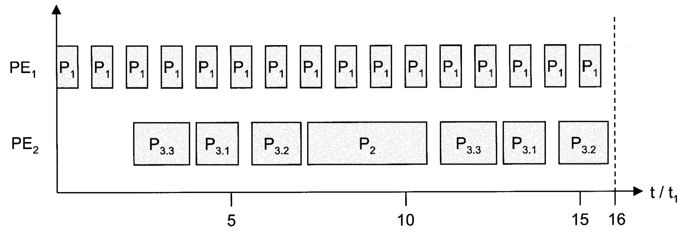


Fig. 7. Gantt diagram of remote motor controller.

Furthermore, the correct periodic execution of $P_{\rm time}$ and $P_{\rm sensor}$ is enforced by the latency path constraints $LC_{(P_{\rm time} \rightarrow Q_{11} \rightarrow P_{\rm time})} = [t_1, t_1]$ and $LC_{(P_{\rm sensor} \rightarrow Q_{14} \rightarrow P_{\rm sensor})} = [t_3, t_3]$, respectively.

For the application of an offline periodic scheduling method, the periods T_p of the remote motor controller processes can be derived from the SPI representation (Fig. 6). This is trivial for processes P_1 and P_2 which are specified as periodic processes (e.g., $T_{P_1} = t_1$) but could also be derived from the SPI representation [e.g., $T_{P_2} = (r_{Q_9}/s_{Q_9})\, T_{P_{\rm time}} = 16t_1$]. Similarly, the periods of the SDF processes $P_{3.1}$, $P_{3.2}$, and $P_{3.3}$ depend on the period of the signal source driving them modeled by process P_{sensor}. Due to the uniform data rates throughout the SDF part, their periods evaluate to $T_{P_{3.i}} = T_{P_{\rm sensor}} = t_3$. Furthermore, the queues between the SDF processes form precedence constraints for their execution. These precedence constraints can be formulated as $P_{3.1}^k \mapsto P_{3.2}^k$ denoting that the kth execution of $P_{3.1}$ has to be completed before $P_{3.2}$ can be executed for the kth time. Similarly holds $P_{3.3}^k \mapsto P_{3.1}^k$ and $P_{3.2}^{(k+1)} \mapsto P_{3.3}^k$ where $(k+1)$ originates from the preassigned token on queue Q_{16}.

Assuming a fixed relation $t_3 = 8t_1$, a macro period which is the least common multiple of all process periods can be calculated. Then, an offline schedule can be generated for all process instances in the macro period by a standard approach such as [36]. A Gantt chart for such a possible scheduling of the motor controller on two processing elements is shown in Fig. 7. Since the used scheduling algorithm does not support preemption P_1 needs to be executed on a separate resource in order not to violate its rate constraint.

VI. Conclusion

We proposed a model for the coherent representation of heterogeneously specified embedded systems with respect to scheduling, allocation and performance validation. With this model called SPI, properties like timing, communication behavior and other properties that are important for the above mentioned problems can be represented using parameter intervals. This allows the coherent representation of system parts with different underlying models of computation as well as the modeling of incomplete information and nondeterministic behavior. The approach includes the capability to vary the degree of modeling accuracy using the concepts of parameter intervals and process modes.

Based on the SPI model, the publicly available SPI workbench is currently under development. This workbench will include open extensible data structures representing the SPI model, extractors for several modeling or specification languages, methods for scheduling, allocation, timing analysis and verification, and a user interface that will allow easy integration of model extensions or new methods. The user will be able to perform experiments by embedding his own methods into the infrastructure provided by the workbench.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions.

References

[1] C. A. Valderama, M. Romdhani, and J. M. Daveau *et al.*, *Hardware/Software Co-Design: Prinicples and Practice*. Norwell, MA: Kluwer , 1997, ch. COSMOS: A Transformational Co-design Tool for Multiprocessor Architectures.

[2] Mentor Graphics. (2001) Seamless data sheet. [Online]. Available: http://www.mentorg.com/.

[3] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele, "Combining multiple models of computation for scheduling and allocation," in *Proc. 6th Int. Workshop. Hardware/Software Co-Design (Codes/CASHE '98)*, Seattle, WA, 1998, pp. 9–13.

[4] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele, "Representation of process mode correlation for scheduling," in *Proc. Int. Conf. Computer-Aided Design (ICCAD '98)*, San Jose, CA, Nov. 1998, pp. 54–61.

[5] K. Richter, D. Ziegenbein, R. Ernst, L. Thiele, and J. Teich, "Representation of function variants for embedded system optimization and synthesis," in *Proc. 36th Design Automation Conf. (DAC '99)*, New Orleans, LA, June 1999, pp. 517–522.

[6] R. Ernst, "Codesign of embedded systems: Status and trends," *IEEE Design Test Comput.*, pp. 45–54, Apr. 1998.

[7] E. A. Lee and Th. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, pp. 773–799, May 1995.

[8] Synopsys. (2001) COSSAP data sheet. [Online]. Available: http://www.synopsys.com/.

[9] Cadence. (2001) Signal processing worksystem SPW data sheet. [Online]. Available: http://www.cadence.com/.

[10] D. Harel and A. Naamad, "The STATEMATE semantics of StateCharts," *ACM Trans. Software Eng. Methodology*, vol. 5, no. 4, pp. 293–333, Oct. 1996.

[11] ITU-T, "Recommendation z.100, specification and description language SDL,", 1993.

[12] F. Balarin, P. Giusto, and A. Jurecska *et al.*, *Hardware–Software Co-Design of Embedded Systems: The POLIS Approach*. Norwell, MA: Kluwer , May 1997.

[13] C. Carreras, J. C. Lopez, M. L. Lopez, C. Delgado-Kloos, N. Martinez, and L. Sanchez, "A co-design methodology based on formal specification and high-level estimation," in *Proc. 4th Int. Workshop Hardware/Software Co-Design (Codes/CASHE '96)*, Pittsburgh, PA, Mar. 1996, pp. 28–35.

[14] R. Ernst and Th. Benner, "Communication, constraints and user directives in COSYMA," Institut für DV-Anlagen, Technische Universität Braunschweig, Tech. Rep. CY-94-2, 1994.

[15] J. Zhu, R. Dömer, and D. D. Gajski, "Syntax and semantics of the SpecC language," in *Proc. '97 Synthesis and System Integration Mixed Technology (SASIMI)*, Osaka, Japan, Dec. 1997, pp. 75–82.

[16] B. Lin, "A system design methodology for software/hardware co-development of telecommunication network applications," in *Proc. 33rd Design Automation Conference (DAC '96)*, Las Vegas, NV, June 1996, pp. 672–677.

[17] P. Chou, R. B. Ortega, and G. Borriello, "The chinook hardware/software co-synthesis system," in *Proc. 8th Int. Symp. Syst. Synthesis (ISSS '95)*, Cannes, France, Sept. 1995, pp. 22–25.

[18] C. Weiler, U. Kebschull, and W. Rosenstiel, "C++ base classes for specification, simulation and partitioning of a hardware/software system," in *Proc. VLSI*, Tokyo, Japan, Aug. 1995, pp. 777–784.

[19] Open SystemC Initiative. (2001) SystemC documentation. [Online]. Available: http://www.systemc.org/.

[20] C. Liu and J. Layland, "Scheduling algorithm for multiprogramming in a hard-real-time environment," *J. ACM*, pp. 46–61, 1973.

[21] E. A. Lee, "Modeling concurrent real-time processes using discrete events," *Ann. Software Eng.*, vol. 7, no. 1–4, pp. 25–45, Apr. 1999.

[22] Synopsys. (2001) CoCentric System Studio data sheet. [Online]. Available: http://www.synopsys.com/.

[23] T. Grötker, R. Schoenen, and H. Meyr, “PCC: A modeling technique for mixed control/data flow systems,” in *Proc. European Design. Test Conf. (ED&TC ’97)*, Paris, France, Mar. 1997, pp. 482–486.

[24] L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich, “FunState—An internal design representation for codesign,” in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD ’99)*, San Jose, CA, Nov. 1999, pp. 558–565.

[25] R. Ernst and A. A. Jerraya, “Embedded system design with multiple languages,” in *Proc. Asia South Pacific Design Automation Conf. (ASPDAC ’00)*, Yokohama, Japan, Jan. 2000, pp. 391–396.

[26] K. Richter, “Developing a general model for scheduling of mixed transformative/reactive systems,” M.S. thesis, Institut für DV-Anlagen, TU Braunschweig, Jan. 1998.

[27] M. Jersak, Y. Cai, D. Ziegenbein, and R. Ernst, “A transformational approach to constraint relaxation of a time-driven simulation model,” in *Proc. 13th Int. Symp. Syst. Synthesis*, Madrid, Spain, Sept. <<AUTHOR: PAGE NO?>> 2000.

[28] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and J. Teich, “Scheduling hardware/software systems using symbolic techniques,” in *Proc. 7th Int. Workshop on Hardware/Software Co-Design (Codes/CASHE ’99)*, Rome, Italy, May 1999, pp. 173–177.

[29] D. Ziegenbein, J. Uerpmann, and R. Ernst, “Dynamic response time optimization for SDF graphs,” in *Proc. Int. Conf. Computer-Aided Design (ICCAD ’00)*, San Jose, Nov. 2000, pp. 135–140.

[30] The MathWorks. (2001) Real-time workshop data sheet. [Online]. Available: http://www.mathworks.com.

[31] D. Ziegenbein, F. Wolf, K. Richter, M. Jersak, and R. Ernst, “Interval-based analysis of software processes,” in *Proc. Workshop Languages, Compilers, Tools Embedded Syst. (LCTES ’01)*, Snowbird, UT, June 2001, pp. 94–101.

[32] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. Comput.*, vol. C–36, Jan. 1987.

[33] J. Teich, T. Blickle, and L. Thiele, “An evolutionary approach to system-level synthesis,” in *Proc. Fifth Int. Workshop. Hardware/Software Co-Design (Codes/CASHE ’97)*, Braunschweig, Germany, Mar. 1997, pp. 167–171.

[34] F. Wolf and R. Ernst, “Execution cost interval refinement in static software analysis,” *J. Syst. Architecture, The EUROMICRO J.* , vol. Special Issue on Modern Methods and Tools in Digital System Design, pp. 339–356, Apr. 2000.

[35] R. Ernst, *System-Level Synthesis*, ser. NATO Science Series. Norwell, MA: Kluwer , 1999, ch. Embedded System Architectures, pp. 1–43.

[36] T. Benner and R. Ernst, “An approach to mixed systems co-synthesis,” in *Proc. 5th Int. Workshop Hardware/Software Co-Design (Codes/CASHE ’97)*, Mar., 1997, pp. 9–14.

Dirk Ziegenbein (M’01) received the M.S. degree in electrical engineering from Virginia Polytechnical University, Blackburg, VA, in 1996.

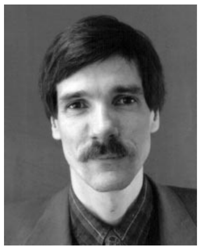
Since 1997, he has been a Member of Rolf Ernst’s Research Group, Institute of Computer and Communication Network Engineering (IDA), Technical University of Braunschweig, Germany, where he is working on the development of the SPI Workbench, an approach to multilanguage embedded system design. His research interests include modeling, analysis, and optimization of complex embedded systems, in particular systems specified using several languages or models of computation.

Kai Richter (M’01) received the diploma (Dipl.-Ing.) degree in electrical engineering from the Technical University of Braunschweig, Germany, in 1998, and is currently working toward the Ph.D degree at the same University.

Since 1998, he has been with Rolf Ernst’s Research Group, Institute of Computer and Communication Network Engineering (IDA), Technical University of Braunschweig, Germany. He is currently involved in the development of performance analysis tools for the SPI Workbench. His research interests include real-time systems, performance analysis, heterogeneous HW/SW platforms, and modeling languages.

Rolf Ernst (M’89) received the diploma in Computer Science and the Ph.D. degree in electrical engineering from the University of Erlangen–Nuremberg, Germany, in 1981 and 1988, respectively.

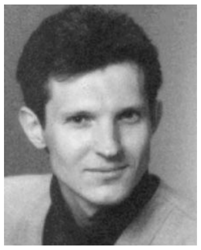
From 1988 to 1989, he was a Member of the Technical Staff with Bell Labs in Allentown, PA. Since 1990, has been a Full Professor with the Technical University of Braunschweig, Germany, and head of the Institute of Computer and Communication Network Engineering (IDA). He is codeveloper of COSYMA, one of the first hardware/software cosynthesis systems. His main research interests include embedded system design and embedded system design automation.



Lothar Thiele (SM’83–M’85) received the Dipl.-Ing. and Dr.-Ing. degrees in electrical engineering from the Technical University of Munich, Germany, in 1981 and 1985, respectively.

From 1981 to 1987, he was a Research Associate with the Institute of Network Theory and Circuit Design, Technical University of Munich, Germany. In 1987, he joined the Information Systems Laboratory, Stanford University, CA. In 1988, he joined the Faculty of Engineering, Saarland University, Saarbrücken, Germany, as Chair of Microelectronics. In 1994, he joined the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, as a Full Professor in Computer Engineering. His research interests include models, methods, and software tools for the design of embedded systems.

In 1986, Dr. Thiele received the Award of The Technical University of Munich for his Ph.D. thesis. He received the 1987 Outstanding Young Author Award of the IEEE Circuits and Systems Society. In 1988, he was the recipient of the 1988 Browder J. Thompson Memorial Prize of the IEEE.



Jürgen Teich (SM’89–M’95) received the Dipl.-Ing. degree (with honors) from the University of Kaiserslautern, Germany and the Ph.D. degree (*summa cum laude*) from Saarland University, Saarbrücken, Germany, in 1989 and 1993, respectively.

In 1994, he joined the DSP design group of Edward A. Lee and David G. Messerschmitt at the University of California Berkeley, where he worked on the Ptolemy Project. From 1995 to 1998, he was with the TIK at the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland. In 1998, he became a Full Professor with the Electrical Engineering and Information Technology Department, and Chair of the Department of Computer Engineering, University of Paderborn, Germany. He is author of *Digitale Hardware/Software-Systeme* (Germany: Springer, 1997). His research interests include massive parallelism, embedded systems, hardware/software codesign, and computer architecture.