4

Models of Embedded Computation

4.1 Introduction **4**-1
Models of Sequential and Parallel Computation • Nonfunctional Properties • Heterogeneity • Component Interaction • Time • The Purpose of an MoC
4.2 The MoC Framework **4**-9
Processes and Signals • Signal Partitioning • Untimed MoCs • The Synchronous MoC • Discrete Timed MoCs
4.3 Integration of MoCs **4**-16
MoC Interfaces • Interface Refinement • MoC Refinement
4.4 Conclusion **4**-22
References **4**-23

Axel Jantsch
Royal Institute of Technology

4.1 Introduction

A model of computation (MoC) is an abstraction of a real computing device. Different computational models serve different objectives and purposes. Thus, they always suppress some properties and details that are irrelevant for the purpose at hand, and they focus on other properties that are essential. Consequently, MoCs have been evolving during the history of computing. In the early decades, between 1920 and 1950, the main focus has been on the question: "What is computable?" The Turing machine and the lambda calculus are prominent examples of computational models developed to investigate that question.[1] It turned out that several, very different MoCs, such as the Turing machine, the lambda calculus, partial recursive functions, register machines, Markov algorithms, Post systems, etc., [1] are all equivalent in the sense that they all denote the same set of computable mathematical functions. Thus, today the so-called Church–Turing thesis is widely accepted:

Church–Turing thesis. If function f is effectively calculable, then f is Turing-computable. If function f is not Turing-computable, then f is not effectively calculable [1, p. 379].

It is the basis for our understanding today what kind of problems can be solved by computers, and what kind of problems principally are beyond a computer's reach. A famous example of what cannot be solved by a computer is the halting problem for Turing machines. A practical consequence is that there cannot be an

[1] The term "model of computation" came in use only much later in the 1970s, but conceptually the computational models of today can certainly be traced back to the models developed in the 1930s.

algorithm that given a function f and a C++ program P (or a program in any other sufficiently complex programming language), could determine if P computes f. This illustrates the principal difficulty of programming language teachers in correcting exams, and of verification engineers in validation programs and circuits.

Later the focus changed to the question: "What can be computed in reasonable time and with reasonable resources?," which spun off the theories of algorithmic complexity based on computational models exposing timing behavior in a particular but abstract way. This resulted in a hierarchy of complexity classes for algorithms according to their *asymptotic complexity*. The computation time (or other resources) for an algorithm is expressed as a function of some characteristic figure of the input, for example, the size of the input. For instance we can state that the function $f(n) = 2n$, for natural numbers n can be computed in $p(n)$ time steps by any computer for some polynomial function $p(n)$. In contrast, the function $g(n) = n!$ cannot be computed in $p(n)$ time steps on any sequential computer for any polynomial function $p(n)$ and arbitrary n. With growing n the time steps required to compute $g(n)$ grows faster than can be expressed by any polynomial function.

This notion of asymptotic complexity allows us to express properties about algorithms in general disregarding details of the algorithms and the computer architecture. This comes at the cost of accuracy. We may only know that there exists some polynomial function $p(n)$ for every computer, but we do not know $p(n)$ since it may be very different for different computers. To be more accurate one needs to take into account more details of the computer architecture. As a matter of fact the complexity theories rest on the assumption that one kind of computational model, or machine abstraction, can simulate another one with a bounded and well-defined overhead. This simulation capability has been expressed in the thesis given below:

Invariance thesis. "Reasonable" machines can simulate each other with a polynomially bounded overhead in time and a constant overhead in space [2].

This thesis establishes an equivalence between different machine models and makes results for a particular machine more generally useful. However, some machines are equipped with considerably more resources and cannot be simulated by a conventional Turing machine according to the invariance thesis. Parallel machines have been the subject of a huge research effort and the question, how parallel resources increase the computational power of a machine has lead to a refinement of computational models and an accuracy increase for estimating computation time. The fundamental relation between sequential and parallel machines has been captured by the following thesis.

Parallel computation thesis. Whatever can be solved in polynomially bounded space on a reasonable sequential machine model, can be solved in polynomially bounded time on a reasonable parallel machine, and vice versa [2].

Parallel computers prompted researchers to refine computational models to include the delay of communication and memory access, which we review briefly in Section 4.1.1.

Embedded systems require a further evolution of computational models due to new design and analysis objectives and constraints. The term "embedded" triggers two important associations. First, an embedded component is squeezed into a bigger system, which implies constraints on size, the form factor, weight, power consumption, cost, etc. Second, it is surrounded by real-world components, which implies timing constraints and interfaces to various communication links, sensors, and actuators. As a consequence, the computational models that are used and are useful in embedded system design are different from those in general purpose sequential and parallel computing. The difference comes from the nonfunctional requirements and constraints and from the heterogeneity.

4.1.1 Models of Sequential and Parallel Computation

Arguably, general purpose sequential computing had for a long time a privileged position, in that it had a single, very simple, and effective MoC. Based on the van Neumann machine, the

random access machine (RAM) *model* [3] is a sufficiently general model to express all important algorithms and reflects the salient nonfunctional characteristics of practical computing engines. Thus, it can be used to analyze performance properties of algorithms in a hardware architecture and implementation independent way. This favorable situation for sequential computing has been eroded over the years as processor architectures and memory hierarchies became ever more complex and deviated from the ideal RAM model.

The parallel computation community has been searching in vain for a similar simple and effective model [4]. Without a universal model of parallel computation, the foundations for the development of portable and efficient parallel applications and architectures were lacking. Consequently, parallel computing has not gained as wide acceptance as sequential computing and is still confined to niche markets and applications. The *parallel random access machine* (PRAM) [5] is perhaps the most popular model of parallel computation and closest to its sequential counterpart with respect to simplicity. A number of processors execute in a lock-step way, that is, synchronized after each cycle governed by a global clock, and access global, shared memory simultaneously within one cycle. The PRAM model's main virtue is its simplicity but it poorly captures the costs associated with computing. Although the RAM model has a similar cost model, there is a significant difference. In the RAM model the costs (execution time, program size) are in fact well reflected and grow linearly with the size of the program and the length of the execution path. This correlation is in principle correct for all sequential processors. The PRAM model does not exhibit this simple correlation because in most parallel computers the cost of memory access, communication and synchronization can be vastly different depending on which memory location is accessed and which processors communicate. Thus, the developer of parallel algorithms does not have sufficient information from the PRAM model alone to develop efficient algorithms. He or she has to consult the specific cost models of the target machine.

Many PRAM variants have been developed to more realistically reflect real cost. Some made the memory access more realistic. The exclusive read–exclusive write (EREW) and the concurrent read–exclusive write (CREW) models [5] serialize access to a given memory location by different processors but still maintain the unit cost model for memory access. The local memory PRAM (LPRAM) model [6] introduces a notion of memory hierarchy while the queued read–queued write (QRQW) PRAM [7] models the latency and contention of memory access. A host of other PRAM variants have factored in the cost of synchronization, communication latency, and bandwidth. Other models of parallel computation, many of which are not directly derived from the PRAM machine, focus on memory. There either the distributed nature of memory is the main concern [8] or the various cost factors of the memory hierarchy are captured [6,9,10]. An introductory survey of models of parallel computation has been written by Maggs et al. [4].

4.1.2 Nonfunctional Properties

A main difference between sequential computation and parallel computation comes from the role of time. In sequential computing, time is solely a performance issue which is moreover captured fairly well by the simple and elegant RAM model. In parallel computing the execution time can only be captured by complex cost functions that depend heavily on various details of the parallel computer. In addition, the execution time can also alter the functional behavior, because the changes in the relative timing of different processors and the communication network can alter the overall functional behavior. To counter this danger, different parts of the parallel program must be synchronized properly.

In embedded systems the situation is even more delicate if real-time deadlines have to be observed. A system that responds slightly too late may be as unacceptable as a system that responds with incorrect data. Even worse, it is entirely context dependent if it is better to respond slightly too late, incorrectly, or not at all. For instance when transmitting a video stream, incorrect data arriving on time may be preferable to correct data arriving too late. Moreover, it may be better not to send data that arrive too late to save resources. On the other hand, control signals to drive the engine or the breaks in a car must always arrive and a tiny delay may be preferable to no signal at all. These observations lead to the distinction of different kinds of real-time systems, for example, hard versus soft real-time systems, depending on the requirements on the timing.

Since most embedded systems interact with real-world objects they are subject to some kind of real-time requirements. Thus, time is an integral part of the functional behavior and cannot be abstracted away completely in many cases. So it should not come as a surprise that MoCs have been developed to allow the modeling of time in an abstract way to meet the application requirements while at the same time avoiding the unnecessary burden of too detailed timing. We will discuss some of these models below. In fact, the timing abstractions of different MoCs is a main organizing principle in this chapter.

Designing for low power is a high priority for most, if not all, embedded systems. However, power has been treated in a limited way in computational models because of the difficulty to abstract the power consumption from the details of architecture and implementation. For very large-scale integration (VLSI) circuits computational models have been developed to derive lower and upper bounds with respect to complexity measures that usually include both circuit area and computation time for a given behavior. AT^2 has been found to be a relevant and interesting complexity measure, where A is the circuit area and T is the computation time either in clock cycles or in physical time. These models have also been used to derive bounds on the energy consumption by usually assuming that the consumed energy is proportional to the state changes of the switching elements. Such analysis shows for instance that AT^2 optimal circuits, that is, circuits which are optimal up to a constant factor with respect to the AT^2 measure for a given boolean function, utilize their resources to a high degree, which means that on average a constant fraction of the chip changes state. Intuitively this is obvious since, if large parts of a circuit are not active over a long period (do not change state), it can presumably be improved by making it either smaller or faster and thus utilizing the circuit resources to a higher degree on average. Or, to conclude the other way round, an AT^2 optimal circuit is also optimal with respect to energy consumption for computing a given boolean function. One can spread out the consumed energy over a larger area or a longer time period, but one cannot decrease the asymptotic energy consumption for computing a given function. Note, that all these results are asymptotic complexity measures with respect to a particular size metric of the computation, for example, the length in bit of the input parameter of the function. For a detailed survey of this theory see Reference 11.

These models have several limitations. They make assumptions about the technology. For instance, in different technologies the correlation between state switching and energy consumption is different. In n-channel metal oxide semiconductor (NMOS) technologies the energy consumption is more correlated with the number of switching elements. The same is true for complementary metal oxide semiconductor (CMOS) technologies if leakage power dominates the overall energy consumption. Also, they provide asymptotic complexity measures for very regular and systematic implementation styles and technologies with a number of assumptions and constraints. However, they do not expose relevant properties for complex modern microprocessors, VLIW (Very Large Instruction Word) processors, DSPs (Digital Signal Processings), FPGAs (Field Programmable Gate Arrays), or ASIC (Application Specific Integrated Circuit) designs in a way useful for system level design decisions. And we are again back at our original question about what exactly is the purpose of a computational model and how general or how specific should it be.

In principle, there are two alternatives to integrate nonfunctional properties such as power, reliability, and also time in an MoC:

- First, we can include these properties in the computational model and associate every functional operation with a specific quantity of that property. For example, an add operation takes 2 nsec and consumes 60 pW. During simulation or some other analysis we can calculate the overall delay and power consumption.
- Second, we can allocate abstract budgets for all parts of the design. For instance, in synchronous design styles, we divide the time axis in slots or cycles and assign every part of the design to exactly one slot. Later on during implementation, we have to find the physical time duration of each slot, which determines the clock frequency. We can optimize for high clock frequency by identifying the critical path and optimizing that design part aggressively. Alternatively, we can move some of the functionality from the slot with the critical part to a neighboring slot, thus balancing the different slots. This budget approach can also be used for managing power consumption, noise, and other properties.

The first approach suffers from inefficient modeling and simulation when all implementation details are included in a model. Also, it cannot be applied to abstract models since there these implementation details are not available. Recall, that a main idea of computational models is that they should be abstract and general enough to support analysis of a large variety of architectures. The inclusion of detailed timing and power consumption data would obstruct this objective. Even the approach to start out with an abstract model and later on back-annotate the detailed data from realistic architectural or implementation models does not help, because the abstract model does not allow to draw concrete conclusions and the detailed, back-annotated model is valid only for a specific architecture.

The second approach with abstract budgets is slightly more appealing to us. On the assumption that all implementations will be able to meet the budgeted constraints, we can draw general conclusions about performance or power consumption on an abstract level valid for a large number of different architectures. One drawback is that we do not know exactly for which class of architectures our analysis is valid, since it is hard to predict which implementations will at the end be able to meet the budget constraints. Another complication is, that we do not know the exact physical size of these budgets and it may indeed be different for different architectures and implementations. For instance an ASIC implementation of a given architecture may be able to meet a cycle constraint of 1 nsec and run at 1 GHz clock frequency, while an FPGA implementation of exactly the same algorithms requires a cycle budget of 5 nsec. But still, the abstract budget approach is promising because it divides the overall problem into more manageable pieces. At the higher level we make assumptions about abstract budgets and analyze a system based on these assumptions. Our analysis will then be valid for all architectures and implementations that meet the stated assumptions. At the lower level we have to ensure and verify that these assumptions are indeed met.

4.1.3 Heterogeneity

Another salient feature of many embedded systems is heterogeneity. It comes from various environmental constraints on the interfaces, from heterogeneous applications and from the need to find different tradeoffs among performance, cost, power consumption, and flexibility for different parts of the system. Consequently, we see analog and mixed signal parts, digital signal processing parts, image, and video processing parts, control parts, and user interfaces coexisting in the same system or even on the same VLSI device. We also see irregular architectures with microprocessors, DSPs, VLIWs, custom hardware coprocessors, memories, and FPGAs connected via a number of different segmented and hierarchical interconnection schemes. It is a formidable task to develop a uniform MoC that exposes all relevant properties while nicely suppressing irrelevant details.

Heterogeneous MoCs is one way to address heterogeneity at the application, architecture, and implementation level. Different computational models are connected and integrated into a hierarchical, heterogeneous MoC that represents the entire system. Many different approaches have been taken to either connect two different computational models or provide a general framework to integrate a number of different models. It turns out that issues of communication, synchronization, and time representation pose the most formidable challenges. The reason is that the communication, and, in particular, the synchronization semantics between different MoC domains correlates the time representation between the two domains. As we will see below, connecting a timed MoC with an untimed model leads to the import of a time structure from the timed to the untimed model resulting in a heterogeneous, timed MoC. Thus the integration cannot stop superficially at the interfaces leaving the interior of the two computational domains unaffected.

Due to the inherent heterogeneity of embedded systems, different MoCs will continue to be used and thus different MoC domains will coexist within the same system. There are two main possible relations, one is due to refinement and the other due to partitioning. A more abstract MoC can be refined into a more detailed model. In our framework, time is the natural parameter that determines the abstraction level of a model. The untimed MoC is more abstract then the synchronous MoC, which in turn is more abstract than the timed MoC. It is in fact common practice that a signal processing algorithm is first

modeled as an untimed dataflow algorithm, which is then refined into a synchronous circuit description, which in turn is mapped onto a technology dependent netlist of fully timed gates.

However, this is not a natural flow for all applications. Control dominated systems or subsystems require some notion of time already at the system level and sensor and actuator subsystems may require a continuous time model right from the start. Thus, different subsystems should be modeled with different MoCs.

4.1.4 Component Interaction

A troubling issue in complex, heterogeneous systems is unexpected behavior of the system due to subtle and complex ways of interaction of different MoCs parts. Eker et al. [12] call this phenomenon *emergent behavior*. Some examples shall illustrate this important point:

Priority inversion. Threads in a real-time operating system may use two different mechanism of resource allocation[12]. One is based on priority and preemption to schedule the threads. The second is based on monitors. Both are well defined and predictable in isolation. For instance, priority and preemption based scheduling means that a higher priority thread cannot be blocked by a lower priority thread. However, if the two threads also use a monitor lock, the lower priority thread may block the high priority thread via the monitor for an indefinite amount of time.

Performance inversion. Assume there are four CPUs on a bus. CPU_1 sends data to CPU_2, CPU_3 sends data to CPU_4 over the bus [13]. We would expect that the overall system performance improves when we replace one CPU with a faster processor, or at least that the system performance does not decrease. However, replacing CPU_1 with a faster CPU_1' may mean that data is sent from CPU_1' to CPU_2 with a higher frequency, at least for a limited amount of time. This means, that the bus is more loaded by this traffic, which may slow down the communication from CPU_3 to CPU_4. If this communication performance has a direct influence on the system performance, we will see a decreased overall system performance.

Over synchronization. Assume that the upper and lower branches in Figure 4.1 have no mutual functional dependence as the dataflow arrows indicate. Assume further that process B is blocked when it tries to send data to C1 or D1, but the receiver is not ready to accept the data. Then, a delay or deadlock in branch D will propagate back through process B to both A and the entire C branch.

These examples are not limited to situations when different MoCs interact. They show that, when separate, seemingly unrelated subsystems interact via a nonobvious mechanism, which is often a shared resource, the effects can be hard to analyze. When the different subsystems are modeled in different MoCs the problem is even more pronounced and harder to analyze due to different communication semantics, synchronization mechanisms, and time representation.

4.1.5 Time

The treatment of time will serve for us as the most important dimension to distinguish MoCs. We can identify at least four levels of accuracy, which are continuous time, discrete time, clocked time, and causality. In the sequel, we only cover the last three levels.

When time is not modeled explicitly, events are only partially ordered with respect to their causal dependences. In one approach, taken for instance in deterministic dataflow networks [14,15], the system

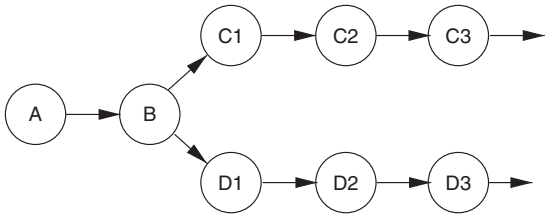


FIGURE 4.1 Over synchronization between functionally independent subsystems.

behavior is independent of delays and timing behavior of computation elements and communication channels. These models are robust with respect to time variations in that any implementation, no matter how slow or fast it is, will exhibit the same behavior as the model. Alternatively, different delays may affect the system's behavior and we obtain an inherently nondeterministic model since time behavior, which is not modeled explicitly is allowed to influence the observable behavior. This approach has been taken both in the context of dataflow models [16–19] and process algebras [20, 21]. In this chapter we follow the deterministic approach, which however can be generalized to approximate nondeterministic behavior by means of stochastic processes as shown in Reference 22.

To exploit the very regular timing of some applications, the synchronous dataflow (SDF) [23] has been developed. Every process consumes and emits a statically fixed number of events in each evaluation cycle. The evaluation cycle is the reference time. The regularity of the application is translated into a restriction of the model, which in turn allows efficient analysis and synthesis techniques that are not applicable for more general models. Scheduling, buffer size optimization, and synthesis has been successfully developed for the SDF.

One facet related to the representation of time is the dichotomy of dataflow dominated and control flow dominated applications. Dataflow dominated applications tend to have events that occur in very regular intervals. Thus, explicit representation of time is not necessary and in fact often inefficient. In contrast, control dominated applications deal with events occurring at very irregular time instants. Consequently, explicit representation of time is a necessity because the timing of events cannot be inferred. Difficulties arise in systems that contain both elements. Unfortunately, these kind of systems become more common since the average system complexity steadily increases. As a consequence, several attempts to integrate dataflow and control dominated modeling concepts have emerged.

In the synchronous piggybacked dataflow model [24] control events are transported on dataflow streams to represent a global state without breaking the locality principle of dataflow models.

The composite signal flow [25] distinguishes between control and dataflow processes and puts significant effort to maintain the frame-oriented processing which is so common in dataflow and signal processing applications for efficiency reasons. However, conflicts occur when irregular control events must be synchronized with dataflow events inside frames. The composite signal flow addresses this problem by allowing an approximation of the synchronization and defines conditions when approximations are safe and do not lead to erroneous behavior.

Time is divided up into time slots or clock cycles by various synchronous models. According to the perfect synchrony assumption [26, 27] neither communication nor computation takes any noticeable time and the time slots or evaluation cycles are completely determined by the arrival of input events. This assumption is useful because designer and tools can concentrate solely on the functionality of the system without mixing this activity with timing considerations. Optimization of performance can be done in a separate step by means of static timing analysis and local retiming techniques. Even though timing does not appear explicitly in synchronous models, the behavior is not independent of time. The model constrains all implementations such that they must be fast enough to process input events properly and to complete an evaluation cycle before the next events arrive. When no events occur in an evaluation cycle, a special token called *absent event* is used to communicate the advance of time. In our framework we use the same technique in Sections 4.2.4 and 4.2.5 for both the synchronous MoC and the fully timed MoC.

Discrete timed models use a discrete set, usually integers or natural numbers, to assign a time stamp to each event. Many discrete event models fall into this category [28–30] as well as most popular hardware description languages, such as VHDL and Verilog. Timing behavior can be modeled most accurately, which makes it the most general model we consider here and makes it applicable to problems such as detailed performance simulation where synchronous and untimed models cannot be used. The price for this is the intimate dependence of functional behavior on timing details and significantly higher computation costs for analysis, simulation, and synthesis problems. Discrete timed models may be nondeterministic, as mainly used in performance analysis and simulation (see e.g., Reference 30), or deterministic, as more desirable for hardware description languages such as VHDL.

The integration of these different timing models into a single framework is a difficult task. Many attempts have been made on a practical level with a concrete design task, mostly simulation, in mind [31–35]. On a conceptual level Lee and Sangiovanni-Vincentelli [36] have proposed a tagged time model in which every event is assigned a time tag. Depending on the tag domain we obtain different MoCs. If the tag domain is a partially ordered set, it results in an untimed model according to our definition. Discrete, totally ordered sets lead to timed MoCs and continuous sets result in continuous time MoCs. There are two main differences between the tagged time model and our proposed framework. First, in the tagged time model processes do not know how much time has progressed when no events are received since global time is only communicated via the time stamps of ordinary events. For instance, a process cannot trigger a time-out if it has not received events for a particular amount of time. Our timed model in Section 4.2.5 does not use time tags but absent events to globally order events. Since absent events are communicated between processes whenever no other event occurs, processes are always informed about the advance of global time. We chose this approach because it resembles better the situation in design languages, such as VHDL, C, or SDL (Specification and Description Language) where processes always can experience time-outs. Second, one of our main motivations was the separation of communication and synchronization issues from the computation part of processes. Hence, we strictly distinguish between process interfaces and process functionality. Only the interfaces determine to which MoC a process belongs, while the core functionality is independent of the MoC. This feature is absent from the tagged token model. This separation of concerns has been inspired by the concept of firing cycles in dataflow process networks [37]. Our mechanism for consuming and emitting events based on signal partitionings as described in Sections 4.2.2 and 4.2.3.1 is only slightly more general than the firing rules described by Lee [37] but it allows a useful definition of process signatures based on the way processes consume and emit events.

4.1.6 The Purpose of an MoC

As mentioned several times, the purpose of a computational model determines, how it is designed, what properties it exposes, and what properties it suppresses.

We argue that MoCs for embedded systems should not address principal questions of computability or feasibility, but should rather aid the design and validation of concrete systems. How this is accomplished best remains a subject of debate, but for this chapter we assume that an MoC should support the following properties:

Implementation independence. An abstract model should not expose too much details of a possible implementation, for example, which kind of processor is used, how much parallel resources are available, what kind of hardware implementation technology is used, details of the memory architecture, etc. Since an MoC is a machine abstraction, it should, by definition, avoid unnecessary machine details. Practically speaking, the benefits of an abstract model include that analysis and processing is faster and more efficient, that analysis results are relevant for a larger set of implementations, and that the same abstract model can be directed to different architectures and implementations. On the downside we note diminished analysis accuracy and a lack of knowledge of the target architecture that can be exploited for modeling and design. Hence, the right abstraction level is a fine line that is also changing over time. While many embedded system designers could for long safely assume a purely sequential implementation, current and future computational models should avoid such an assumption. Resource sharing and scheduling strategies become more complex, and an MoC should thus either allow the explicit modeling of such a strategy or restrict the implementations to follow a particular, well-defined strategy.

Composability. Since many parts and components are typically developed independently and integrated into a system, it is important to avoid unexpected interferences. Thus, some kind of *composability property* [38] is desirable. One step in this direction is to have a deterministic computational model such as Kahn process networks that guarantee a particular behavior independent of the time or individual activities and independent of the amount of available resources in general.

This is of course only a first step since, as argued earlier, time behavior is often an integral part of the functional behavior. Thus, resource sharing strategies, that greatly influence timing, will still have a major impact on the system behavior even for fully deterministic models. We can reconcile good system composability with shared resources by allocating a minimum but guaranteed amount of resources for each subsystem or task. For instance, two tasks get a fixed share of the communication bandwidth of a bus. This approach allows for ideal composability but has to be based on worst case behavior. It is very conservative and hence, does not utilize resources efficiently.

We can relax this approach by allocating abstract resource budgets as part of the computational model. Then we require from the implementation to provide the requested resources, and at the same time to minimize the abstract budgets and thus the required resources. As example consider two tasks that have a particular communication need per abstract time slot, where the communication need may be different for different slots. The implementation has to fulfill the communication requirements of all tasks by providing the necessary bandwidth in each time slot, tuning the length of the individual time slots, or by moving communication from one slot to another. These optimizations will also have to consider global timing and resource constraints. In any case, in the abstract model we can deal with abstract budgets and assume that they will be provided by any valid implementation.

Analyzability. A general tradeoff exists between the expressiveness of a model and its analyzability. By restricting models in clever ways, one can apply powerful and efficient analysis and synthesis methods. For instance, the SDF model allows all actors only a constant amount of input and output tokens in each activation cycle. While this restricts the expressiveness of the model, it allows to efficiently compute static schedules when they exist. For general dataflow graphs this may not be possible because it could be impossible to ensure that the amount of input and output is always constant for all actors, even if they are in a particular case. Since SDF covers a fairly large and important application domain, it has become a very useful MoC. The key is to understand what are the important properties (finding static schedules, finding memory bounds, finding maximum delays, etc.) and devising an MoC that allows to handle these properties efficiently and does not restrict the modeling power too much.

In the following sections we discuss a framework to study different MoCs. The idea is to use different types of process constructors to instantiate processes of different MoCs. Thus, one type of process constructors would yield only untimed processes, while another type results in timed processes. The elements for process construction are simple functions and are in principle independent of a particular MoC. However, the independence is not complete since some MoCs put specific constraints on the functions. But still the separation of the process interfaces from the internal process behavior is fairly far reaching. The interfaces determine the time representation, synchronization, and communication, hence the MoC.

In this chapter we will not elaborate all interesting and desirable properties of computational models. Rather we will use the framework to introduce four different MoCs that only differ in their timing abstraction. Since time plays a very prominent role in embedded systems, we focus on this aspect and show how different time abstractions can serve different purposes and needs. Another defining aspect of embedded systems is heterogeneity, which we address by allowing different MoCs to coexist in a model. The common framework makes this integration semantically clean and simple. We study two particular aspects of this coexistence, namely the interfaces between two different MoCs and the refinement of one MoC into another.

Other central issues of embedded systems, such as power consumption, global analysis and optimization, are not covered, mostly because they are not very well understood in this context and few advanced proposals exist on how to deal with them from an MoC perspective.

4.2 The MoC Framework

In the remainder of this chapter we discuss a framework that accommodates MoCs with different timing abstractions. It is based on *process constructor*, which is a mechanism to instantiate processes. A process constructor takes one or more pure functions as arguments and creates a process. The functions represent

the process behavior and have no notion of time or concurrency. They simply take arguments and produce results. The process constructor is responsible for establishing communication with other processes. It defines the time representation, the communication, and synchronization semantics. A set of process constructors determines a particular MoC. This leads to a systematic and clean separation of computation and communication. A function, that defines the computation of a process, can in principle be used to instantiate processes in different computational models. However, a computational model may put constraints on functions. For instance, the synchronous MoC requires a function to take exactly one event on each input and produce exactly one event for each output. The untimed MoC does not have a similar requirement.

After some preliminary definitions in this section, we introduce the untimed processes, give a formal definition of an MoC, and define the untimed MoC (Section 4.2.3) the perfectly synchronous and the clocked synchronous MoC (Section 4.2.4), and the discrete time MoC (Section 4.2.5). Based on this we introduce interfaces between MoCs and present an interface refinement procedure in the next section. Furthermore, we discuss the refinement from an untimed MoC to a synchronous MoC and to a timed MoC.

4.2.1 Processes and Signals

Processes communicate with each other by writing to and reading from signals. Given is a set of values V, which represents the data communicated over the signals. *Events*, which are the basic elements of signals, are or contain values. We distinguish among three different kinds of events.

Untimed events $\dot{E}$ are just values without further information, $\dot{E} = V$. *Synchronous events* $\bar{E}$ include a pseudo-value $\perp$ in addition to the normal values, hence $\bar{E} = V \cup \{\perp\}$. *Timed events* $\hat{E}$ are identical to synchronous events, $\hat{E} = \bar{E}$. However, since it is often useful to distinguish them, we use different symbols. Intuitively, timed events occur at much finer granularity than synchronous events and they would usually represent physical time units, such as a nanosecond. In contrast, synchronous events represent abstract time slots or clock cycles. This model of events and time can only accommodate discrete time models. Continuous time would require a different representation of time and events. We use the symbols $\dot{e}$, $\bar{e}$, and $\hat{e}$ to denote individual untimed, synchronous, and timed events, respectively. We use $E = \dot{E} \cup \bar{E} \cup \hat{E}$ and $e \in E$ to denote any kind of event.

Signals are sequences of events. Sequences are ordered and we use subscripts as in e_i to denote the ith event in a signal. For example, a signal may be written as $\langle e_0, e_1, e_2 \rangle$. In general, signals can be finite or infinite sequences of events and S is the set of all signals. We also distinguish among three kinds of signals and $\dot{S}$, $\bar{S}$, and $\hat{S}$ denote the untimed, synchronous, and timed signal sets, respectively, and $\dot{s}$, $\bar{s}$, and $\hat{s}$ designate individual untimed, synchronous, and timed signals.

$\langle\rangle$ is the empty signal and $\oplus$ concatenates two signals. Concatenation is associative and has the empty signal as its neutral element: $s_1 \oplus (s_2 \oplus s_3) = (s_1 \oplus s_2) \oplus s_3$, $\langle\rangle \oplus s = s \oplus \langle\rangle = s$. To keep the notation simple we often treat individual events as one-event sequences, for example, we may write $e \oplus s$ to denote $\langle e \rangle \oplus s$.

We use angle brackets, "$\langle$" and "$\rangle$"not only to denote ordered sets or sequences of events, but also to denote sequences of signals if we impose an order on a set of signals.

$\#s$ gives the length of signal s. Infinite signals have infinite length and $\#\langle\rangle = 0$.
[] is an index operation to extract an event on a particular position from a signal.

For example, $s[2] = e_2$ if $s = \langle e_1, e_2, e_3 \rangle$.

Processes are defined as functions on signals

$$p : S \to S.$$

Processes are functions in the sense that for a given input signal we always get the same output signal, that is, $s = s' \Rightarrow p(s) = p(s')$. Note, that this still allows processes to have an internal state. Thus, a process does not necessarily react identical to the same event applied at different times. But it will

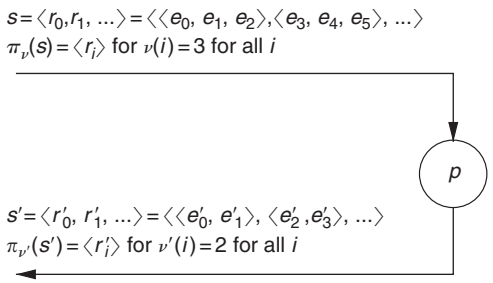


FIGURE 4.2 The input signal of process p is partitioned into an infinite sequence of subsignals each of which contains three events, while the output signal is partitioned into subsignals of lengths 2.

produce the same, possibly infinite, output signal when confronted with identical, possibly infinite, input signals provided it starts with the same initial state.

4.2.2 Signal Partitioning

We shall use the partitioning of signals into subsequences to define the portions of a signal that is consumed or emitted by a process in each evaluation cycle.

A *partition* $\pi(\nu, s)$ of a signal s defines an ordered set of signals, $\langle r_i \rangle$, which, when concatenated together, form "almost" the original signal s. The function $\nu : \mathbb{N}_0 \to \mathbb{N}_0$ defines the lengths of all elements in the partition. $\nu(0) = \#r_0$ gives the length of the first element in the partition, $\nu(1) = \#r_1$ gives the length of the second element, etc.

Example 4.1 Let $s_1 = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$ and $\nu_1(0) = \nu_1(1) = 3, \nu_1(2) = 4$. Then we get the partition $\pi(\nu_1, s_1) = \langle \langle 1, 2, 3 \rangle, \langle 4, 5, 6 \rangle, \langle 7, 8, 9, 10 \rangle \rangle$.

Let $s_2 = \langle 1, 2, 3, \ldots \rangle$ be the infinite signal with ascending integers. Let $\nu_2(i) = 2$ for all $i \geq 0$. The resulting partition is infinite: $\pi(\nu_2, s_2) = \langle \langle 1, 2 \rangle, \langle 3, 4 \rangle, \ldots \rangle$.

The function $\nu(i)$ defines the length of the subsignals r_i. If it is constant for all i we usually omit the argument and write ν. Figure 4.2 illustrates a process with an input signal s and an output signal s'. s is partitioned into subsignals of length 3 and s' into subsignals of length 2.

4.2.3 Untimed MoCs

4.2.3.1 Process Constructors

Our aim is to relate functions of events to processes, which are functions of signals. Therefore we introduce *process constructors* that can be considered as higher-order functions. They take functions on events as arguments and return processes. We define only a few basic process constructors that can be used to compose more complex processes and process networks. All untimed process constructors and processes operate exclusively on untimed signals.

Processes with arbitrary number of input and output signals are cumbersome to deal with in a formal way. To avoid this inconvenience we mostly deal with processes with one input and one output. To handle arbitrary processes also, we introduce "zip" and "unzip" processes that merge two input signals into one and split one input signal into two output signals, respectively. These processes together with appropriate process composition allows us to express arbitrary behavior.

Processes instantiated with the `mealyU` constructor resemble Mealy state machines in that they have a next state function and an output encoding function that depend on both the input and the current state.

Definition 4.1 *Let V be an arbitrary set of values, let $g, f : (V \times \dot{S}) \to \dot{S}$ the next state and output encoding functions, let $\gamma : V \to \mathbb{N}$ be a function defining the input partitioning, and let $w_0 \in V$ be an initial state. `mealyU` is a process constructor which, given γ, f, g, and w_0 as arguments, instantiates a process $p : \dot{S} \to \dot{S}$.*

The function γ determines the number of events consumed by the process in the current evaluation cycle. γ is dependent on the current state. p repeatedly applies g on the current state and the input events to compute the next state. Further, it applies f repeatedly on the current state and the input events, to compute the output events.

Processes instantiated by `mealyU` are general state machines with one input and one output. To create processes with arbitrary inputs and outputs, we also use the following constructors:

- `zipU` instantiates a process with two inputs and one output. In every evaluation cycle this process takes one event from the left input and one event from the right input and packs them into an event pair that is emitted at the output.
- `unzipU` instantiates a process with one input and two outputs. In every evaluation cycle this process takes one event from the input. It requires it to be an event pair. The first event of this pair is emitted to the left output, the second event of the event pair is emitted to the right output.

For truly general process networks we would in fact need more complex zip processes, but for the purpose of this chapter the simple constructors are sufficient and we refer the reader for details to Reference 39.

4.2.3.2 Composition Operators

We consider only three basic composition operators, namely sequential composition, parallel composition, and feedback.

We give the definitions only for processes with one or two input and output signals, because the generalization to arbitrary numbers of inputs and outputs is straightforward.

Definition 4.2 *Let $p_1, p_2 : \dot{S} \to \dot{S}$ be two processes with one input and one output each, and let $s_1, s_2 \in \dot{S}$ be two signals. Their parallel composition, denoted as $p_1 \parallel p_2$, is defined as follows.*

$$(p_1 \parallel p_2)(\langle s_1, s_2 \rangle) = \langle p_1(s_1), p_2(s_2) \rangle.$$

Since processes are functions we can easily define sequential composition in terms of functional composition.

Definition 4.3 *Let again $p_1, p_2 : \dot{S} \to \dot{S}$ be two processes and let $s \in \dot{S}$ be a signal. The sequential composition, denoted as $p_1 \circ p_2$, is defined as follows.*

$$(p_2 \circ p_1)(s) = p_2(p_1(s)).$$

Definition 4.4 *Given a process $p : (S \times S) \to (S \times S)$ with two input signals and two output signals we define the process $\mu p : S \to S$ by the equation*

$$(\mu p)(s_1) = s_2 \quad \textit{where } p(s_1, s_3) = (s_2, s_3).$$

The behavior of the process μp is defined by the least fixed point semantics based on the prefix order of signals.

The μ operator gives feedback loops (Figure 4.3) a well-defined semantics. Moreover, the value of the feedback signal can be constructed by repeatedly simulating the process network starting with the empty signal until the values on all feedback signals stabilize and do not change any more [39].

Now we are in a position to define precisely what we mean with an MoC.

Definition 4.5 *An MoC is a 2-tuple* $\mathrm{MoC} = (C, O)$*, where C is a set of process constructors, each of which, when given constructor specific parameters, instantiates a process. O is a set of process composition operators, each of which, when given processes as arguments, instantiates a new process.*

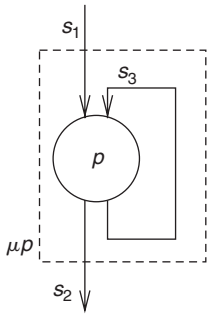


FIGURE 4.3 Feedback composition of a process.

Definition 4.6 *The* untimed MoC *is defined as untimed* MoC $= (C, O)$*, where*

$$C = \{\texttt{mealyU}, \texttt{zipU}, \texttt{unzipU}\}$$
$$O = \{\|, \circ, \mu\}.$$

In other words, a process or a process network belongs to the untimed MoC domain *iff all its processes and process compositions are constructed either by one of the named process constructors or by one of the composition operators. We call such processes* U-MoC processes.

Because the process interface is separated from the functionality of the process, interesting transformations can be done. For instance, a process can be mechanically transformed into a process that consumes and produces a multiple number of events of the original process. Processes can be easily merged into more complex processes. Moreover, there may be the opportunity to move functionality from one process to another. For more details on this kind of transformations see Reference 39.

4.2.4 The Synchronous MoC

The synchronous languages StateCharts [40], Esterel [41], Signal [42], Argos, Lustre [43], and some others have been developed on the basis of the perfect synchrony assumption.

Perfect synchrony hypothesis. *Neither computation nor communication takes time.*

Timing is entirely determined by the arriving of input events because the system processes input samples in zero time and then waits until the next input arrives. If the implementation of the system is fast enough to process all inputs before the next sample arrives, it will behave exactly as the specification in the synchronous language.

4.2.4.1 Process Constructors

Formally, we develop synchronous processes as a special case of untimed processes. This will allow us later to easily connect different domains.

Synchronous processes have two specific characteristics. First, all synchronous processes consume and produce exactly one event on each input or output in each evaluation cycle, that is, the signature is always $\langle\{1, \ldots\}, \{1, \ldots\}\rangle$. Second, in addition to the value set V events can carry the special value $\perp$, which denotes the *absence* of an event; this is the way we defined synchronous events $\bar{E}$ and signals $\bar{S}$ in Section 4.2.1. Both, the processes and their contained functions must be able to deal with these events.

All synchronous process constructors and processes operate exclusively on synchronous signals.

Definition 4.7 *Let V be an arbitrary set of values, $\bar{E} = V \cup \{\perp\}$, let $g, f : (\bar{E} \times \bar{S}) \to \bar{S}$ and let $w_0 \in V$ be an initial state.* `mealyS` *is a process constructor which, given f, g, and w_0 as arguments, instantiates a process $p : \bar{S} \to \bar{S}$. p repeatedly applies g on the current state and the input event to compute the next state. Further it*

applies repeatedly f on the current state and the input event to compute the output event. p consumes exactly one input event in each evaluation cycle and emits exactly one output event.

We only require that g and f are defined for absent input events and that the output signal partitioning is the constant 1.

When we merge two signals into one we have to decide how to represent the absence of an event in one input signal in the compound signal. We choose to use the $\perp$ symbol for this purpose also, which has the consequence, that $\perp$ appears also in tuples together with normal values. Thus, it is essentially used for two different purposes. Having clarified this, the definition for `zipS` and `unzipS` is straightforward. `zipS`-based processes pack two events from the two inputs into an event pair at the output, while `unzipS` performs the inverse operation.

4.2.4.2 The Perfectly Synchronous MoC

Again, we can now make precise what we mean by synchronous MoC.

Definition 4.8 *The* synchronous MoC *is defined as synchronous* MoC $= (C, O)$, *where*

$$C = \{\texttt{mealyS}, \texttt{zipS}, \texttt{unzipS}\}$$
$$O = \{\|, \circ, \mu_S\}.$$

In other words, a process or a process network belongs to the synchronous MoC domain *iff all its processes and process compositions are constructed either by one of the named process constructors or by one of the composition operators. We call such processes* S-MoC processes.

Note, that we do not use the same feedback operator for the synchronous MoC. μ_S defines the semantics of the feedback loop based on the Scott order of the values in $\bar{E}$. It is also based on a fixed point semantics but it is resolved *for each event* and not over a complete signal. We have adopted μ_S to be consistent with the zero-delay feedback loop semantics of most synchronous languages. For our purpose here this is not significant and we do not need to go into more details. For precise definitions and a thorough motivation the reader is referred to Reference 39.

Merging of processes and other related transformations are very simple in the synchronous MoC because all processes have essentially identical interfaces. For instance, the merge of two `mealyS`-based processes can be formulated as follows.

$$\texttt{mealyS}(g_1, f_1, v_0) \circ \texttt{mealyS}(g_2, f_2, w_0) = \texttt{mealyS}(g, f, (v_0, w_0))$$
$$\text{where } g((v, w), \bar{e}) = (g_1(v, f_2(w, \bar{e})), g_2(w, \bar{e})) \quad f((v, w), \bar{e}) = f_1(v, f_2(w, \bar{e})).$$

4.2.4.3 The Clocked Synchronous MoC

It is useful to define a variant of the perfectly synchronous MoC, the clocked synchronous MoC that is based on the following hypothesis.

Clocked synchronous hypothesis. *There is a global clock signal controlling the start of each computation in the system. Communication takes no time and computation takes one clock cycle.*

First, we define a delay process Δ that delays all inputs by one evaluation cycle.

$$\Delta = \texttt{mealyS}(f, g, \perp)$$
$$\text{where } g(w, \bar{e}) = \bar{e}, \quad f(w, \bar{e}) = w.$$

Based on this delay process we define the constructors for the clocked synchronous model.

Definition 4.9

$$\begin{aligned} \texttt{mealyCS}(g,f,w_0) &= \texttt{mealyS}(g,f,w_0) \circ \Delta \\ \texttt{zipCS}()(\bar{s}_1,\bar{s}_2) &= \texttt{zipS}()(\Delta(\bar{s}_1),\Delta(\bar{s}_2)) \\ \texttt{unzipCS}() &= \texttt{unzipS}() \circ \Delta. \end{aligned} \tag{4.1}$$

Thus, elementary processes are composed of a combinatorial function and a delay function that essentially represents a latch at the inputs.

Definition 4.10 *The* clocked synchronous MoC *is defined as clocked synchronous* MoC $= (C, O)$*, where*

$$C = \{\texttt{mealyCS}, \texttt{zipCS}, \texttt{unzipCS}\}$$
$$O = \{\|, \circ, \mu\}.$$

In other words, a process or a process network belongs to the clocked synchronous MoC Domain *iff all its processes and process compositions are constructed either by one of the named process constructors or by one of the composition operators. We call such processes* CS-MoC processes.

4.2.5 Discrete Timed MoCs

Timed processes are a blend of untimed and synchronous processes in that they can consume and produce more than one event per cycle and they also deal with absent events. In addition, they have to comply with the constraint that output events cannot occur before the input events of the same evaluation cycle. This is achieved by enforcing an equal number of input and output events for each evaluation cycle, and by prepending an initial sequence of absent events. Since the signals also represent the progression of time, the prefix of absent events at the outputs corresponds to an initial delay of the process in reacting to the inputs. Moreover, the partitioning of input and output signals corresponds to the duration of each evaluation cycle.

Definition 4.11 *`mealyT` is a process constructor which, given γ, f, g, and w_0 as arguments, instantiates a process $p : \hat{S} \to \hat{S}$. Again, γ is a function of the current state and determines the number of input events consumed in a particular evaluation cycle. Function g computes the next state and f computes the output events with the constraint that the output events do not occur earlier than the input events on which they depend.*

This constraint is necessary because in the timed MoC each event corresponds to a time stamp and we have a globally total order of time, relating all events in all signals to each other. To avoid causality flaws every process has to abide by this constraint.

Similarly `zipT`-based processes consume events from their two inputs and pack them into tuples of events emitted at the output. `unzipT` performs the inverse operation. Both have also to comply with the causality constraint.

Again, we can now make precise what we mean by timed MoC.

Definition 4.12 *The* timed MoC *is defined as timed* MoC $= (C, O)$*, where*

$$C = \{\texttt{mealyT}, \texttt{zipT}, \texttt{unzipT}\}$$
$$O = \{\|, \circ, \mu\}.$$

In other words, a process or a process network belongs to the timed MoC domain *iff all its processes and process compositions are constructed either by one of the named process constructors or by one of the composition operators. We call such processes* T-MoC processes.

Merging other transformations as well as analysis of time process networks is more complicated than for synchronous or untimed MoCs, because the timing may interfere with the pure functional behavior. However, we can further restrict the functions used in constructing the processes, to more or less separate behavior from timing also in the timed MoC. To illustrate this we discuss a few variants of the Mealy process constructor.

`mealyPT`. In `mealyPT` (γ, f, g, w_0) based processes the functions f and g are not exposed to absent events and they are only defined on untimed sequences. The interface of the process strips-off all absent events of the input signal, hands over the result to f and g, and inserts absent events at the output as appropriate to provide proper timing for the output signal. The function γ, which may depend on the process state as usual, defines how many events are consumed. Essentially, it represents a timer and determines when the input should be checked the next time.

`mealyST`. In `mealyST` (γ, f, g, w_0) based processes γ determines the number of nonabsent events that should be handed over to f and g for processing. Again, f and g never see or produce absent events and the process interface is responsible for providing them with the appropriate input data and for synchronization and timing issues on inputs and outputs. Unlike `mealyPT` processes, functions f and g in `mealyST` processes have no influence on when they are invoked. They only control how many nonabsent events have appeared before their invocation. f and g in `mealyPT` processes on the other hand determine the time instant of their next invocation independent of the number of nonabsent events.

`mealyTT`. However, a combination of these two process constructors is `mealyTT`, which allows to control the number of nonabsent input events and a maximum time period, after which the process is activated in any case independent of the number of nonabsent input events received. This allows to model processes that wait for input events but can set internal timers to provide time-outs.

These examples illustrate that process constructors and MoCs could be defined, which allow us to precisely define to which extent communication issues are separated from the purely functional behavior of the processes. Obviously, a stricter separation greatly facilitates verification and synthesis but may restrict expressiveness.

4.3 Integration of MoCs

4.3.1 MoC Interfaces

Interfaces between different MoCs determine the relation of the time structure in the different domains and they influence the way a domain is triggered to evaluate inputs and produce outputs. If an MoC domain is time triggered, the time signal is made available through the interface. Other domains are triggered when input data is available. Again, the input data appears through the interfaces.

We introduce a few simple interfaces for the MoCs of the previous sections, in order to be able to discuss concrete examples.

Definition 4.13 *A `stripS2U` process constructor takes no arguments and instantiates a process* $p : \bar{S} \to \dot{S}$*, which takes a synchronous signal as input and generates an untimed signal as output. It reproduces all data from the input in the output in the same order with the exception of the absent event, which is translated into the value 0.*

Definition 4.14 *An `insertU2S` process constructor takes no arguments and instantiates a process* $p : \dot{S} \to \bar{S}$*, which takes an untimed signal as input and generates a synchronous signal as output. It reproduces all data from the input in the output in the same order without any change.*

These interface processes between the synchronous and the untimed MoCs are very simple. However, they establish a strict and explicit time relation between two connected domains.

Connecting processes from different MoCs also requires a proper semantic basis, which we provide by defining a hierarchical MoC.

Definition 4.15 *A hierarchical model of computation (HMoC) is a 3-tuple HMoC* $= (M, C, O)$*, where* M *is a set of HMoCs or simple MoCs, each capable of instantiating processes or process networks;* C *is a set of process constructors;* O *is a set of process composition operators that governs the process composition at the highest hierarchy level but not inside process networks instantiated by any of the HMoCs of* M*.*

In the following examples and discussion we will use a specific but rather simple HMoC.

Definition 4.16 $H = (M, C, O)$ *with*

$$M = \{U\text{-}MoC, S\text{-}MoC\}$$
$$C = \{\texttt{stripS2U}, \texttt{insertU2S}\}$$
$$O = \{\|, \circ, \mu\}.$$

Example 4.2 As example, consider the equalizer system of Figure 4.4 [39]. The control part consists of two synchronous MoC processes and the dataflow part, modeled as untimed MoC processes, filter and analyze an audio stream. Depending on the analysis results of the Analyzer process, the Distortion control will modify the filter parameters. The Button control takes also user input into account to steer the filter. The purpose of Analyzer and Distortion control are to avoid dangerously strong signals that could jeopardize the loud speakers.

Control and dataflow parts are connected via two interface processes. The dataflow processes can be developed and verified separately in the untimed MoC domain, but as soon as they are connected to the synchronous MoC control part, the time structure of the synchronous MoC domain gets imposed on all the untimed MoC processes. With the simple interfaces of Figure 4.4, the Filter process consumes 4096 data tokens from the primary input, 1 token from the `stripS2U` process, and it emits 4096 tokens in every synchronous MoC time slot. Similarly, the activity of the Analyzer is precisely defined for every synchronous MoC time slot. Also, the activities of the two control processes are related precisely to the activities of the dataflow processes in every time slot. Moreover, the timing of the two primary inputs and the primary outputs are now related timewise. Their timing must be consistent because the timing of the primary input data determines the timing of the entire system. For example, if the input signal to

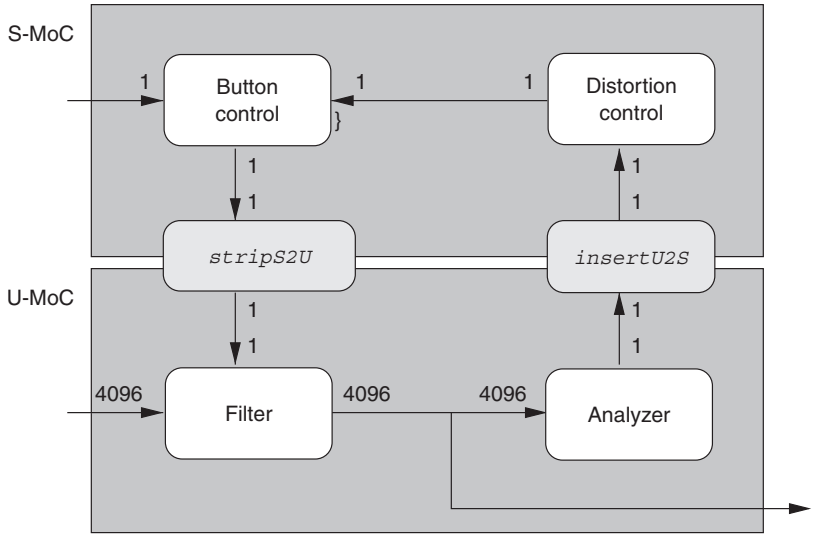


FIGURE 4.4 A digital equalizer consisting of a dataflow part and control. The numbers annotating process inputs and outputs denote the number of tokens consumed and produced in each evaluation cycle. (From A. Jantsch. Modeling Embedded Systems and SoCs. Morgan Kaufmann Publishers, 2004. With permission.)

AQ: Please provide the publisher location for the reference.

the Button control process assumes that each time slot has the same time duration, the 4096 data samples of the Filter input in each evaluation cycle must correspond to the same constant time period. It is the responsibility of the domain interfaces to correctly relate the timing of the different domains to each other. It is required that the time relation established by all interfaces is consistent with each other and with the timing of the primary inputs. For instance if the `stripS2U` takes 1 token as input and emits 1 token as output in each evaluation cycle, the `insertU2S` process cannot take 1 token as input and produce 2 tokens as output.

The interfaces in Figure 4.4 are very simple and lead to a strict coupling between the two MoC domains. Could more sophisticated or nondeterministic interfaces avoid this coupling effect? The answer is no because even if the input and output tokens of the interfaces vary from evaluation cycle to evaluation cycle in complex or nondeterministic ways, we still have a very precise timing relation in each and every time slot. Since in every evaluation cycle all interface processes must consume and produce a particular number of tokens, this determines the time relation in that particular cycle. Even though this relation may vary from cycle to cycle, it is still well defined for all cycles and hence for the entire execution of the system.

The possibly nondeterministic communication delay between MoC domains, as well as between any other processes, can be modeled, but this should not be confused with establishing a time relation between two MoC domains.

4.3.2 Interface Refinement

In order to show this difference and to illustrate how abstract interfaces can be gradually refined to accommodate channel delay information and detailed protocols, we propose an interface refinement procedure, given below:

1. *Add a time interface.* When we connect two different MoC domains, we always have to define the time relation between the two. This is the case even if the two domains are of the same type, for example, both are synchronous MoC domains, because the basic time unit may or may not be identical in the two domains.

In our MoC framework the occurrence of events also represent time in both the synchronous MoC and timed MoC domains. Thus, setting the time relation means to determine the number of events in one domain that correspond to one event in the other domain. For example, in Figure 4.4 the interfaces establish a one-to-one relation while the interface in Figure 4.5 represents a 3/2 relation.

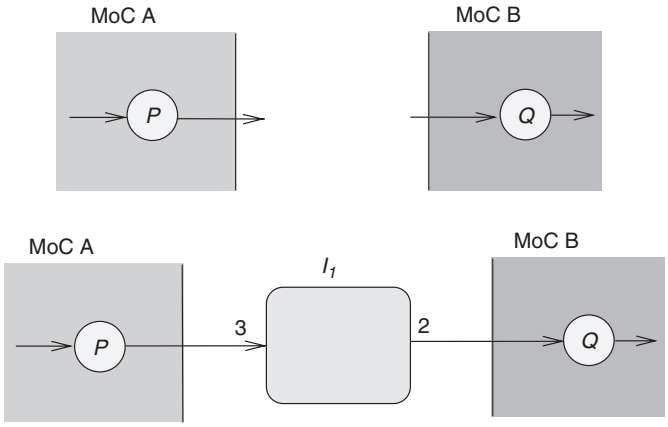


FIGURE 4.5 Determining the time relation between two MoC domains. (From A. Jantsch. Modeling Embedded Systems and SoCs. Morgan Kaufmann Publishers, 2004. With permission.)

In other frameworks the establishing of a time relation will take a different form. For instance, if languages such as SystemC or VHDL are used, the time of the different domains have to be related to the common time base of the simulator.

2. *Refine the protocol.* When the time relation between the two domains is established, we have to provide a protocol that is able to communicate over the final interface at that point. The two domains may represent different clocking regimes on the same chip, or one may end up as software while the other is implemented as hardware, or both may be implemented as software on different chips or cores, etc. Depending on the final implementations we have to develop a protocol fulfilling the requirements of the interface, such as buffering and error control.

In our example in Figure 4.6 we have selected a simple handshake protocol with limited buffering capability. Note, however, that this assumes that for every three events arriving from MoC A there are only two useful events to be delivered to MoC B. The interface processes I_1 and I_2, and the protocol processes P_1, P_2, Q_1, and Q_2 must be designed carefully to avoid both losing data and deadlock.

3. *Model the channel delay.* In order to have a realistic channel behavior, the delay can be modeled deterministically or stochastically. In Figure 4.7 we have added a stochastic delay varying between 2 and 5 MoC B cycles. The protocol will require more buffering to accommodate the varying delays. To dimension the buffers correctly we have to identify the average and the worst-case behavior that we should be able to handle.

This refinement procedure proposed here is consistent with and complementary to other techniques proposed, for example, in the context of SystemC [44]. We only want to emphasize here that the time relation between domains from channel delay and protocol design have to be separated. Often these issues

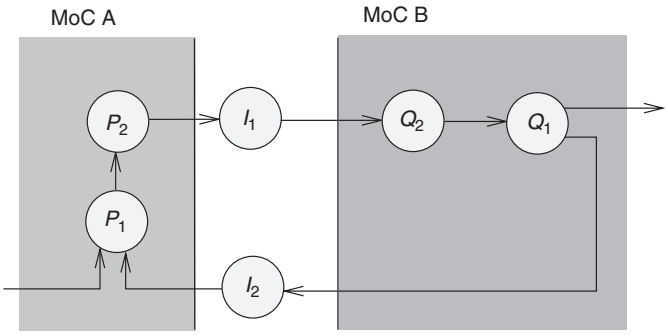


FIGURE 4.6 A simple handshake protocol. (From A. Jantsch. Modeling Embedded Systems and SoCs. Morgan Kaufmann Publishers, 2004. With permission.)

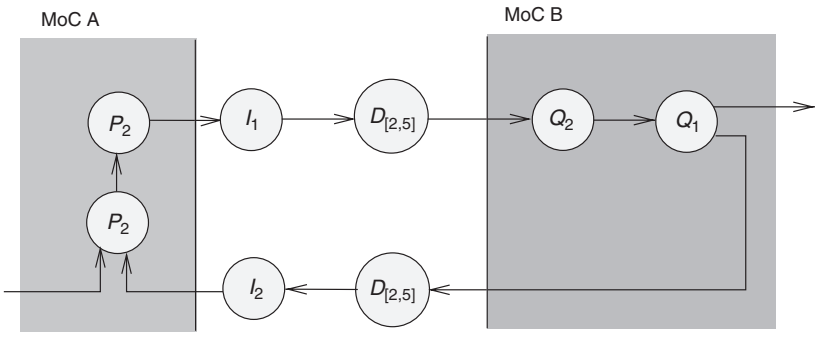


FIGURE 4.7 The channel delay can vary between 2 and 5 cycles measured in MoC B cycles. (From A. Jantsch. Modeling Embedded Systems and SoCs. Morgan Kaufmann Publishers, 2004. With permission.)

are not separated clearly making interface design more complicated than necessary. More details about this procedure and the example can be found in Reference 39.

4.3.3 MoC Refinement

The three introduced MoCs represent three time abstractions and, naturally, design often starts with higher time abstractions and gradually leads to lower abstractions. It is not always appropriate to start with an untimed MoC because when timing properties are an inherent and crucial part of the functionality, a synchronous model is more appropriate to start with. But if we start with an untimed model, we need to map it onto an architecture with concrete timing properties. Frequently, resource sharing makes the consideration of time functionally relevant, because of deadlock problems and complicated interaction patterns. All the three phenomenon discussed in Section 4.1.4, priority inversion, performance inversion, and over-synchronization, emerged due to resource sharing.

Example 4.3 We discuss therefore an example for MoC refinement from the untimed through the synchronous to the timed MoC, which is driven by resource sharing. In Figure 4.8 we have two unlimited MoC process pairs, which are functionally independent from each other. At this level, under the assumption of infinite buffers and unlimited resources, we can analyze and develop the core functionality embodied by the process internal functions f and g.

In the first refinement step, shown in Figure 4.9, we introduce finite buffers between the processes. $B_{n,2}$ and $B_{M,2}$ represent buffers of size n and m, respectively. Since the untimed MoC assumes implicitly infinite buffers between two communicating processes, there is no point in modeling finite buffers in the untimed MoC domain. We just would not see any effect. In the synchronous MoC domain, however, we can analyze

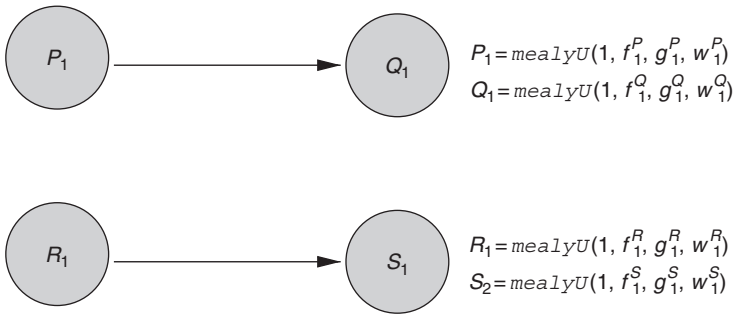


FIGURE 4.8 Two independent process pairs.

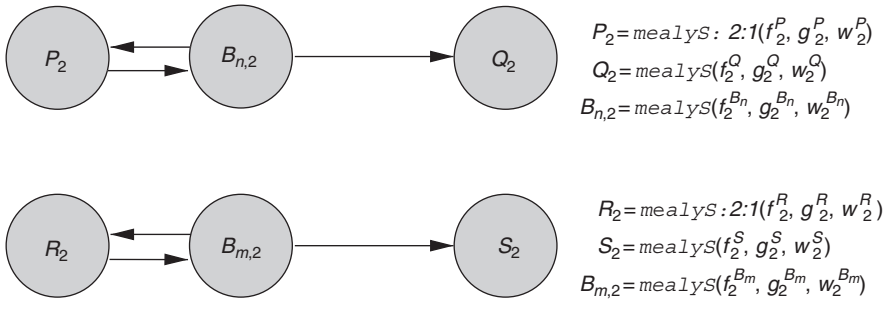


FIGURE 4.9 Two independent process pairs with explicit buffers.

the consequences of finite buffers. The processes need to be refined. Processes P_2 and R_2 have to be able to handle full buffers while processes Q_2 and S_2 have to handle empty buffers. In the untimed MoC, processes always block on empty input buffers. This behavior can also be modeled in synchronous MoC processes easily. In addition more complicated behavior such as time-outs can be modeled and analyzed. To find the minimum buffer sizes while avoiding deadlock and ensuring the original system behavior is by itself a challenging task. Basten and Hoogerbrugge [45] propose a technique to address this. More frequently, the buffer minimization problem is formulated as part of the process scheduling problem [46,47].

The communication infrastructure is typically shared among many communicating actors. In Figure 4.10 we map the communication links onto one bus, represented as process I_3. It contains an arbiter that resolves conflicts when both processes $B_{n,3}$ and $B_{m,3}$ try to access the bus at the same time. It also implements a bus access protocol, that has to be followed by connecting processes. The synchronous MoC model in Figure 4.10 is cycle true and the effect of bus sharing on system behavior and performance can be analyzed. A model checker can use the soundness and fairness of the arbitration algorithm and performance requirements on the individual processes can be derived to achieve a desirable system performance.

Sometimes, it is a feasible option to synthesize the model of Figure 4.10 directly into a hardware or software implementation, provided we can use standard templates for the process interfaces. Alternatively we can refine the model into a fully timed model. However, we still have various options depending on what exactly we would like to model and analyze. For each process we can decide how much of the timing and synchronization details should be explicitly taken care of by the process and how much can be handled implicitly by the process interfaces. For instance in Section 4.2.5 we have introduced constructors `mealyST` and `mealyPT`. The first provides a process interface that strips-off all absent events and inserts absent events at the output as needed. The internal functions have only to deal with the functional events but they have no access to timing information. This means that an untimed `mealyU` process can be directly refined into a timed `mealyST` process with exactly the same functions f and g. Alternatively, the constructor `mealyPT` provides an interface that invokes the internal functions at regular time intervals. If this interval corresponds to a synchronous time slot, a synchronous MoC process can be easily mapped onto a `mealyPT` type of process, with the only difference, that the functions in a `mealyPT` process may receive several nonabsent events in each cycle. But in both cases the processes experience a notion of time based on cycles.

In Figure 4.11 we have chosen to refine processes P, Q, R, and S into `mealyST`-based processes to keep them as similar to the original untimed processes. Thus, the original f and g functions can be used without major modification. The process interfaces are responsible to collect the inputs, present them to the f and g functions and emit properly synchronized output.

The buffer and the bus processes however have been mapped onto `mealyPT` processes. The constants λ and $\lambda/2$ represent the cycle time for the processes. Process $B_{m,4}$ operates with half the cycle time of

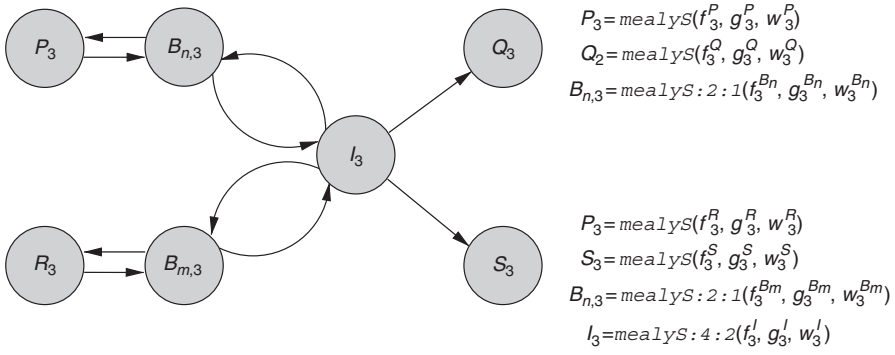
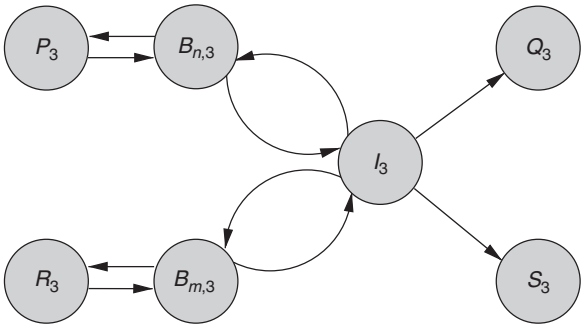


FIGURE 4.10 Two independent process pairs with explicit buffers.



$$P_4 = \texttt{mealyST}(1, f_4^P, g_4^P, w_4^P)$$
$$Q_2 = \texttt{mealyST}(f_4^Q, g_4^Q, w_4^Q)$$
$$B_{n,4} = \texttt{mealyPT:2:1}(\lambda, f_4^{Bn}, g_4^{Bn}, w_4^{Bn})$$

$$R_4 = \texttt{mealyST}(1, f_4^R, g_4^R, w_4^R)$$
$$S_4 = \texttt{mealyS}(1, f_4^S, g_4^S, w_4^S)$$
$$B_{m,4} = \texttt{mealyPT:2:1}(\frac{\lambda}{2}, f_4^{Bm}, g_4^{Bm}, w_4^{Bm})$$

$$I_4 = \texttt{mealyPT:4:2}(\lambda, f_4^I, g_4^I, w_4^I)$$

FIGURE 4.11 All processes are refined into the timed MoC but with different synchronization interfaces.

the other processes, which illustrates that the modeling accuracy can be arbitrarily selected. We can also choose other process constructors and hence interfaces if desirable. For instance, some processes can be mapped onto *mealyT*-type processes in a further refinement step to expose them to even more timing information.

4.4 Conclusion

We tried to motivate that MoC for embedded systems should be different from the many computational models developed in the past. The purpose of model of embedded computation should be to support analysis and design of concrete systems. Thus, it needs to deal with salient and critical features of embedded systems in a systematic way. These features include real-time requirements, power consumption, architecture heterogeneity, application heterogeneity, and real-world interaction.

We have proposed a framework to study different MoCs that allow us to appropriately capture some, but unfortunately not all, of these features. In particular power consumption and other non-functional properties are not covered. Time is of central focus in the framework but continuous time models are not included in spite of their relevance for the sensors and actuators in embedded systems.

Despite the deficiencies of this framework we hope that we were able to argue well for a few important points:

- Different computational models should and will continue to coexist for a variety of technical and nontechnical reasons.
- To use the "right" computational model in a design and for a particular design task can greatly facilitate the design process and the quality of the result. What is the "right" model depends on the purpose and objectives of a design task.
- Time is of central importance and computational models with different timing abstractions should be used during system development.

From an MoC perspective, several important issues are open research topics and should be addressed urgently to improve the design process for embedded systems:

- We need to identify efficient ways to capture a few important nonfunctional properties in MoCs. At least power and energy consumption and perhaps signal noise issues should be attended to.
- The effective integration of different MoCs will require (1) the systematic manipulation and refinement of MoC interfaces and interdomain protocols; (2) the crossdomain analysis of functionality, performance, and power consumption; (3) the global optimization and synthesis including migration of tasks and processes across MoC domain boundaries.
- In order to make the benefits and the potential of well-defined MoCs available in the practical design work, we need to project MoCs into design languages, such as VHDL, Verilog, SystemC, C++, etc. This should be done by properly subsetting a language and by developing pragmatics to restrict the use of a language. If accompanied by tools to enforce the restrictions and to exploit the properties of the underlying MoC, this will be accepted quickly by designers.

In the future we foresee a continuous and steady further development of MoCs to match future theoretical objectives and practical design purposes. But we also hope that they become better accepted as practically useful devices for supporting the design process just like design languages, tools, and methodologies.

References

[1] Ralph Gregory Taylor. *Models of Computation and Formal Language*. Oxford University Press, New York, 1998.

[2] Peter van Embde Boas. Machine models and simulation. In J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*. Elsevier Science Publishers B.V., 1990, chap. 1, pp. 1–66.

[3] S. Cook and R. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7:354–375, 1973.

[4] B.M. Maggs, L.R. Matheson, and R.E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, Vol. 2, 1995, pp. 61–70.

[5] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, San Diego, CA, 1978.

[6] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.

[7] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The QRQW PRAM: accounting for contention in parallel algorithms. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, Arlington, VA, January 1994, pp. 638–648.

[8] Eli Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31:507–517, 1984.

[9] A. Aggarwal, B. Alpern, A.K. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, May 1987, pp. 305–314.

[10] Bowen Alpern, Larry Carter, Ephraim Feig, and Ted Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.

[11] Thomas Lengauer. VLSI theory. In J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, 2nd ed., Elsevier Science Publishers, 1990, chap. 16, pp. 835–868.

[12] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity? The ptolemy approach. *Proceedings of the IEEE*, 91:127–144, 2003.

[13] Rolf Ernst. Mpsoc Performance Modeling and Analysis. *Paper Presented at the 3rd International Seminar on Application-Specific Multi-Processor SoC*, Chamonix, France, 2003.

AQ: Please provide place of publication for Refs. [2, 11, 14, 34, and 39].

AQ: Please provide the volume and page number for Ref. [15].

[14] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*. North-Holland, 1974.

[15] Edward A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 1995.

[16] Jarvis Dean Brock. A Formal Model for Non-Deterministic Dataflow Computation. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1983.

[17] J. Dean Brock and William B. Ackerman. Scenarios: a model of nondeterminate computation. In J. Diaz and I. Ramos, Eds., *Formalism of Programming Concepts*, Vol. 107 of *Lecture Notes in Computer Science*. Springer Verlag, Heidelberg, 1981, pp. 252–259.

[18] Paul R. Kosinski. A straight forward denotational semantics for nondeterminate data flow programs. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, 1978, pp. 214–219.

[19] David Park. The 'fairness' problem and nondeterministic computing networks. In J.W. De Baker and J. van Leeuwen, Eds., *Foundations of Computer Science IV, Part 2: Semantics and Logic*. Mathematical Centre Tracts, Amsterdam, The Netherlands, 1983, Vol. 159, pp. 133–161.

[20] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, New York, 1989.

[21] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–676, 1978.

[22] Axel Jantsch, Ingo Sander, and Wenbiao Wu. The usage of stochastic processes in embedded system specifications. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign*, April 2001.

[23] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36:24–35, 1987.

[24] Chanik Park, Jaewoong Jung, and Soonhoi Ha. Extended synchronous dataflow for efficient DSP system prototyping. *Design Automation for Embedded Systems*, 6:295–322, 2002.

[25] Axel Jantsch and Per Bjuréus. Composite signal flow: a computational model combining events, sampled streams, and vectors. In *Proceedings of the Design and Test Europe Conference (DATE)*, 2000.

[26] Nicolas Halbwachs. Synchronous programming of reactive systems. In *Proceedings of Computer Aided Verification (CAV)*, 2000.

[27] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79:1270–1282, 1991.

[28] Frank L. Severance. *System Modeling and Simulation*. John Wiley & Sons, New York, 2001.

[29] Averill M. Law and W. David Kelton. *Simulation, Modeling and Analsysis*, 3rd ed., Industrial Engineering Series. McGraw Hill, New York, 2000.

[30] Christos G. Cassandras. *Discrete Event Systems*. Aksen Associates, Boston, MA, 1993.

[31] Per Bjuréus and Axel Jantsch. Modeling of mixed control and dataflow systems in MASCOT. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9:690–704, 2001.

[32] Peeter Ellervee, Shashi Kumar, Axel Jantsch, Bengt Svantesson, Thomas Meincke, and Ahmed Hemani. IRSYD: an internal representation for heterogeneous embedded systems. In *Proceedings of the 16th NORCHIP Conference*, 1998.

[33] P. Le Marrec, C.A. Valderrama, F. Hessel, A.A. Jerraya, M. Attia, and O. Cayrol. Hardware, software and mechanical cosimulation for auto-motive applications. In *Proceedings of the Ninth International Workshop on Rapid System Prototyping*, 1998, pp. 202–206.

[34] Ahmed A. Jerraya and K. O'Brien. Solar: an intermediate format for system-level modeling and synthesis. In Jerzy Rozenblit and Klaus Buchenrieder, Eds., *Codesign: Computer-Aided Software/Hardware Engineering*. IEEE Press, 1995, chap. 7, pp. 145–175.

[35] Edward A. Lee and David G. Messerschmitt. An Overview of the Ptolemy Project. Report from Department of Electrical Engineering and Computer Science, University of California, Berkeley, January 1993.

[36] Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, 1998.

[37] Edward A. Lee. A Denotational Semantics for Dataflow with Firing. Technical report UCB/ERL M97/3, Department of Electrical Engineering and Computer Science, University of California, Berkeley, January 1997.

[38] Axel Jantsch and Hannu Tenhunen. Will networks on chip close the productivity gap? In Axel Jantsch and Hannu Tenhunen, Eds., *Networks on Chip*, Kluwer Academic Publishers, Dordrecht, 2003, chap. 1, pp. 3–18.

[39] Axel Jantsch. *Modeling Embedded Systems and SoCs — Concurrency and Time in Models of Computation. Systems on Silicon.* Morgan Kaufmann Publishers, 2003.

[40] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[41] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: an introduction to Esterel. In Kazuhiro Fuchi and M. Nivat, Eds., *Programming of Future Generation Computers*, Elsevier, New York, 1988, pp. 35–55.

[42] Paul le Guernic, Thierry Gautier, Michel le Borgne, and Claude le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79:1321–1336, 1991.

[43] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79:1305–1320, 1991.

[44] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC.* Kluwer Academic Publishers, Dordrecht, 2002.

[45] Twan Basten and Jan Hoogerbrugge. Efficient execution of process networks. In Alan Chalmers, Majid Mirmehdi, and Henk Muller, Eds., *Communicating Process Architectures.* IOS Press, 2001.

[46] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, New York, 2000.

[47] Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. *Software Synthesis from Dataflow Graphs.* Kluwer Academic Publishers, Dordrecht, 1996.

AQ: Please provide place of publisher for Ref. [45].