Models of Computation in the Design Process

Axel Jantsch and Ingo Sander
Royal Institute of Technology, Stockholm, Sweden

January 30, 2005

Abstract

We organize Models of Computation (MoC) with respect to their time abstraction. We distinguish between continuous time, discrete time, synchronous and untimed MoCs. System level models serve a variety of objectives with partially contradicting requirements. Consequently, different MoCs are necessary for the various tasks and phases in the design of an embedded system. We trace the impact of MoCs on the efficiency of several design activities for synthesis, verification and simulation.

Contents

1 Introduction

A system on a chip (SoC) can integrate a large number of components such as microcontrollers, digital signal processors (DSPs), memories, custom hardware, and reconfigurable hardware in the form of field programmable gate arrays (FPGAs) together with analog-to-digital (A/D) and digital-to-analog (D/A) converters on a single chip (Figure 1). The communication structures become ever more sophisticated consisting of several connected and segmented buses or even packet switched networks. In

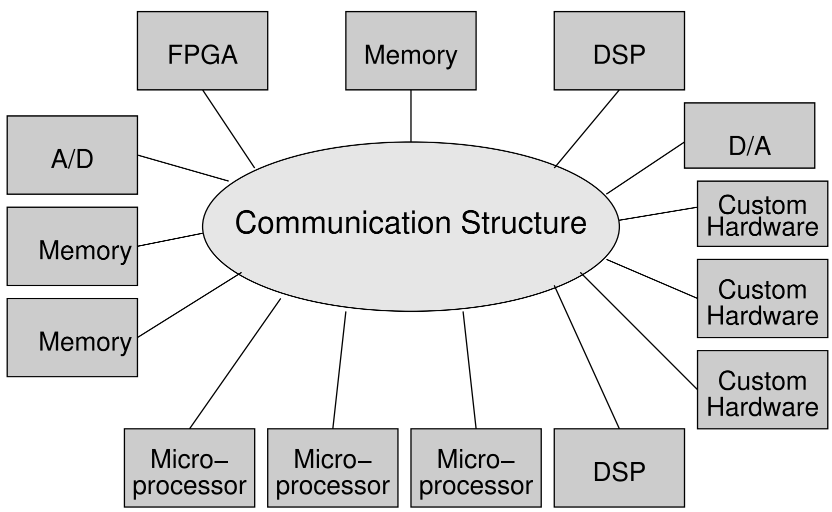


Figure 1: A possible system-on-a-chip architecture

total there may be dozens or hundreds of such components on a single SoC. These architectures offer an enormous potential but they are heterogeneous and tremendously complex. This also applies to embedded software. Moreover, the overall system complexity grows faster than system size due to the component interaction. In fact, intra-system communication is becoming the dominant factor for design, validation and performance analysis. Consequently, issues of communication, synchronization and concurrency must play a prominent role in all system design models and languages.

The design process for SoCs is complex and sophisticated. From abstract models for requirements definition and system specification more and more refined models are derived leading eventually to low level implementation models that describe the layout and the assembler code. Most of the models are generated and processed either fully automatically or with tool support. Once created models have to be verified to check their consistency and correctness.

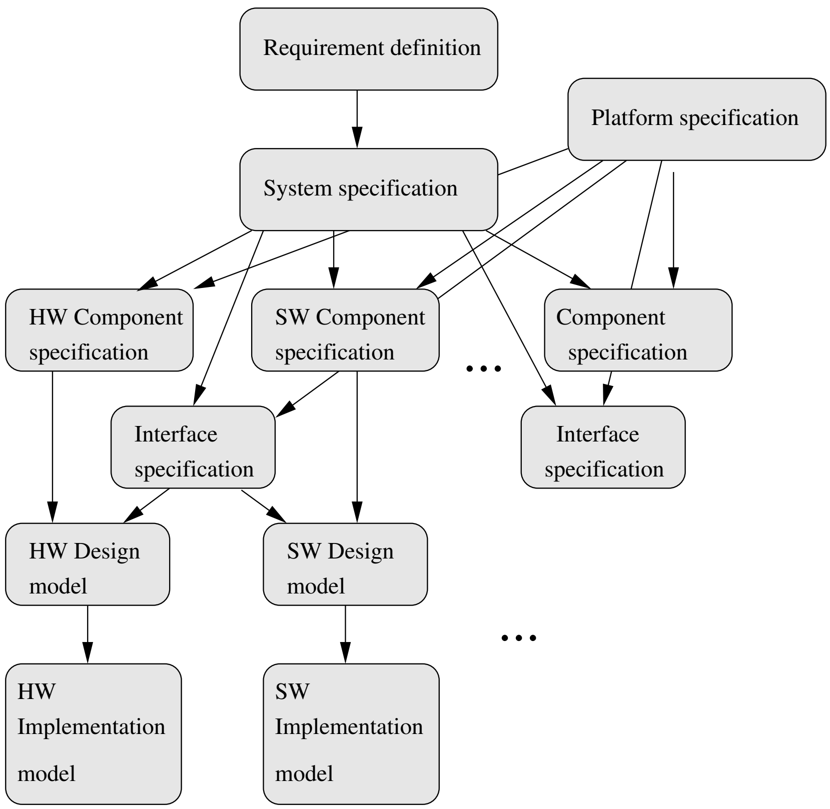


Figure 2: A SoC Design Process involves many models.

Figure 2 depicts a few of the models typically generated and transformed during a design project. Different design tasks require different models. A system level feasibility study and performance analysis needs key performance properties of the architecture, components and functions but not a full behavioral model. Scheduling and schedulability analysis need abstract task graphs, timing requirements and an abstract model of the scheduler in the operating system. Synthesis and verification tools need behavioral models at a proper abstraction level. Noise, EMC analysis, test pattern generators and many other tools have their own requirements on the models they use.

Since all design tasks put specific requirements on a model, we may ask, how strong the influence of a model of computation is on the potential and efficiency of design techniques. The answers are dependent on the specific design tasks and tools. We consider only a small selection of tasks, namely HW synthesis, simulation and formal verification. Also we cannot take all possible models into account, but we restrict the discussion to three specific MoC classes: untimed MoCs, synchronous MoCs, discrete and continuous time MoCs. They are distinguished by their abstraction of time and their synchronization behavior which will allow us to analyze design techniques with respect to these aspects. Other aspects such as data representation will not be covered.

In the next section we introduce the MoCs under consideration and review some of their important properties. In section 3 we trace MoCs in different design phases and in section 4 we discuss the importance of MoCs for synthesis, simulation and verification techniques.

2 Models of Computation

We use the term "Model of Computation" (MoC) to focus on issues of concurrency and time. Consequently, even though it has been defined in different ways by different authors (see for instance [21, 33, 39, 43, 46]), we use it to define the time representation and the semantics of communication and synchronization between processes in a process network. Thus, a MoC defines how computation takes place in a structure of concurrent processes, hence giving a semantics to such a structure [9, 20]. This semantics can be used to formulate an abstract machine that is able to execute a model. *Languages* are not computational models, but have underlying computational models. For instance the languages VHDL, Verilog and SystemC share the same discrete time, event driven computational model. On the other hand, languages can be used to support more than one computational model. In ForSyDe [42] the functional language Haskell [25] is used to express several models of computation. Libraries have been created for synchronous, untimed and discrete time models of computation. Standard ML has been used similarly [35]. SystemC has also been extended to support SDF (synchronous dataflow) and CSP (communicating sequential processes) MoCs in addition to its native discrete time MoC [38].

To choose the right model of computation is of utmost importance, since each MoC has certain properties. As an example consider a process network modeled as a discrete time system in SystemC. In the general case automatic tools will not be able to compute a static schedule for a single processor implementation, even if the process network would easily allow it. For this reason Patel and Shukla [38] have extended SystemC to support an SDF MoC. The same process network expressed as an SDF can then easily be statically scheduled by a tool.

Skillicorn and Talia discuss models of computation for parallel architectures in [44]. Their community faces similar problems as those in design of embedded systems. In fact all typical parallel computer structures (SIMD, MIMD[1]) can be implemented on a SoC architecture. Recognizing, that programming of a large number of communicating processors is an extremely complex task, they try to define properties for a suitable model of parallel computation. They emphasize that a model should hide most of the details (decomposition, mapping, communication, synchronization) from programmers, if they shall be able to manage intellectually the creation of software. The exact structure of the program should be inserted by the translation process rather than by the programmer. Thus models should be as abstract as possible, which means that the parallelism has not even to be made explicit in the program text. They point out that ad hoc compilation techniques cannot be expected to work on problems of this complexity, but advocate building software, that is correct by construction rather then verifying program properties after construction. Programs should be architecture independent to allow reuse. The model should support cost measures to guide the design process and should have guaranteed performance over a useful variety of architectures.

In the following sections, we present a number of important models of computations and give their key properties. Following [20, 21]we organize them according to their time abstraction. We distinguish between discrete time models, synchronous models where a cycle denotes an abstract notion of time, and untimed models. This is consistent with the tagged-signal model proposed by Lee and Sangiovanni-Vincentelli [33]. There each event has a time tag and different time tag structures result in different MoCs. E.g. if the time tags correspond to real numbers we have a continuous time model; integer time tags result in discrete time models; time tags drawn from a partially ordered set result in an untimed MoC.

MoCs can be organized along other criteria, e.g. along the kinds of elements manipulated in a MoC which leads Paul and Thomas [39] to a grouping of MoCs for hardware artifacts, for software artifacts and for design artifacts. However, an organization along properties that are not inherent is of limited use because it changes when MoCs are used in different ways.

A consequence of an organization along the time abstraction is that all strictly sequential models such as finite state machines and sequential algorithms are not distinguished. All of them can serve for modeling individual processes, while the semantics of the MoC defines the process interaction and synchronization.

2.1 Continuous Time Models

When time is represented by a continuous set, usually the real numbers, we talk of a continuous time MoC. Prominent examples of continuous time MoC instances are Simulink [8], VHDL-AMS and Modelica [12]. The behavior is typically expressed

[1]Flynn has classified typical parallel data structures in [13], where SIMD is an abbreviation for Single Instruction, Multiple Data and MIMD for Multiple Instruction, Multiple Data.

as equations over real numbers. Simulators for continuous time MoCs are based on differential equation solvers that compute the behavior of a model including arbitrary internal feed-back loops.

Due to the need to solve differential equations, simulations of continuous time models are very slow. Hence, only small parts of a system are usually modeled with continuous time such as analog and mixed signal components.

To be able to model and analyze a complete system that contains analog components, mixed-signal languages and simulators such as VHDL-AMS have been developed. They allow to model the pure digital parts in a discrete time MoC and the analog parts in a continuous time MoC. This allows for complete system simulations with acceptable simulation performance. It is also a typical example where heterogeneous models based on multiple MoCs have clear benefits.

2.2 Discrete Time Models

Models, where all events are associated with a time instant and the time is represented by a discrete set, such as the natural numbers, are called discrete time models.[2]

Discrete time models are often used for the simulation of hardware. Both VHDL [19] and Verilog [18] use a discrete time model for their simulation semantics. A simulator for discrete time MoCs is usually implemented with a global event queue that sorts occurring events. Discrete time models may have causality problems due to zero-delay in feedback loops, which are discussed in Section 2.4.

2.3 Synchronous Models

In synchronous MoCs time is also represented by a discrete set, but the elementary time unit is not a physical unit but more abstract due to two abstraction mechanisms:

1. Each event occurs in a specific evaluation cycle (also called time slot or clock cycle). The occurrence of evaluation cycles is globally synchronized even for independent parts of the system. But the relative occurrence of events within the same evaluation cycle is not further specified. Thus, events within an evaluation cycle are only partially ordered as defined by causality and data dependences only.

2. Intermediate events that are not visible at the end of an elementary evaluation cycle are irrelevant and can be ignored.

[2]Sometimes this group of MoCs is denoted as *discrete event MoC*. However, strictly speaking "discrete event" and "discrete time" are independent, orthogonal concepts. The first denotes a model where the set of the event values is a discrete set while the second denotes a model with time values drawn from a discrete set, e.g. integers. In contrast *continuous time* and *continuous event* models use continuous sets for time and event values, respectively, e.g. the real numbers. All four combinations occur in practice: continuous time/continuous event models, continuous time/discrete event models, discrete time/continuous event models and discrete time/discrete event models. See for instance [6] for a good coverage of discrete event models.

In each evaluation cycle all processes evaluate once and all events occurring during this process are considered to occur simultaneously.

The synchronous assumption can be formulated according to [1]. The synchronous approach considers "*ideal* reactive systems that produce their outputs *synchronously* with their inputs, their reaction taking no observable time". This implies that the computation of an output event is instantaneous. The synchronous assumption leads to a clean separation between computation and communication. A global clock triggers computations that are conceptually simultaneous and instantaneous. This assumption frees the designer from the modeling of complex communication mechanisms and provides a solid base for formal methods.

A synchronous design technique has been used in hardware design for clocked synchronous circuits. A circuit behavior can be described deterministically independent of the detailed timing of gates by separating combinational blocks from each other with clocked registers. An implementation will have the same behavior as the abstract circuit under the assumption that the combinational blocks are "fast enough" and that the abstract circuit does not include zero-delay feedback loops.

The synchronous assumption implies a simple and formal communication model. Concurrent processes can easily be composed together. However, feedback loops with zero-delay may cause causality problems which are discussed next.

2.4 Feedback Loops in Discrete Time and Synchronous Models

Discrete time models allow zero-delay computation; in perfectly synchronous models this is even a basic assumption. As a consequence, feedback loops may introduce inconsistent behavior. In fact, feedback loops as illustrated in Figure 3 may have no solution, it may have one solution or it may have many solutions.

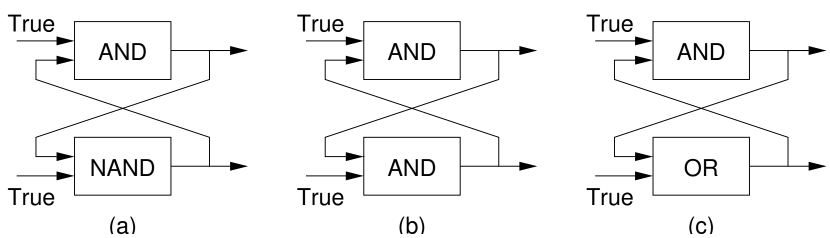


Figure 3: A feedback loop in a synchronous system. System a) has no solutions, b) has multiple solutions and c) has a single solution.

Figure 3a shows a system with a zero-delay feedback loop that does not have a stable solution. If the output of the Boolean AND function is True then the output of the NAND function is False. But this means that the output of the AND function has to be False, which is in contradiction to the starting point of the analysis. Starting with the value False on the output of AND does not lead to a stable solution either. Clearly there is no solution to this problem.

Figure 3b shows a system with feedback loop with multiple solutions. Here the system is stable, if both AND functions have False or if both AND functions have True as their output value. Thus the system has two possible solutions.

Figure 3c shows a feedback loop with only one solution. Here the only solution is that both outputs are True.

It is crucial for the design of safety-critical systems that feedback loops with no solution as in Figure 3a are detected and eliminated, since they result in an oscillator. Also feedback loops with multiple solutions imply a risk for safety-critical systems, since they lead to non-determinism. Non-determinism may be acceptable, if it is detected and the designer is aware of its implications, but may have serious consequences, if it stays undetected.

Since feedback loops in discrete time and synchronous models are of such importance there are several approaches which address this problem [9].

Microstep In order to introduce an order between events that are produced and consumed in an event cycle, the concept of microsteps has been introduced into languages like VHDL. VHDL distinguishes between two dimensions of time. The first one is given by a time unit, e.g. a picosecond, while the second is given by a number of delta-delays. A delta-delay is an infinitesimal small amount of time. An operation may take zero time units, but it takes at least one delta-delay. Delta-delays are used to order operations within the same time unit.

While this approach partly solves the zero-delay feedback problem, it introduces another problem since delta delays will never cause the advance of time measured in time units. Thus during an event cycle there may be an infinite amount of delta-delays. This would be the result, if Figure 3a would be implemented in VHDL, since each operation causes time to advance with one delta-delay. An advantage of the delta-delay is that simulation will reveal that the composite function oscillates. However, a VHDL simulation would not detect that Figure 3b has two solutions, since the simulation semantics of VHDL would assign an initial value for the output of the AND gates (False[3]) and thus would only give one stable solution, concealing the non-determinism from the designer. Another serious drawback of the microstep concept is that it leads to a more complicated semantics, which complicates formal reasoning and synthesis.

Forbid zero-delays The easiest way to cope with the zero-delay feedback problem is to forbid them. In case of Figure 3a and 3b this would mean the insertion

[3] VHDL defines the data type `boolean` by means of `type boolean is (false, true)`. At program start variables and signals take the leftmost value of their data type definitions; in case of the boolean data type the value `false` is used.

of an extra delay function, e.g. after the upper AND function. Since a delay function has an initial value the systems will stabilize. Assuming an initial value of True, Figure 3a will stabilize in the current event cycle with the values False for the output of the NAND function and False for the value of the AND function. Figure 3b would stabilize with the output value True for both AND functions. A possible problem with this approach is that a stable system such as 3c is rejected, since it contains a zero delay feedback loop. This approach is adopted in the synchronous language Lustre [16] and in synchronous digital hardware design. When used in a synchronous MoC this the resulting MoC variant is sometimes called *clocked synchronous MoC* [21].

Unique fixed-point The idea of this approach is that a system is seen as a set of equations for which one solution in form of a fixed-point exists. There is a special value $\perp$ ("bottom") that allows it to give systems with no solution or many solutions a fixed-point solution. The advantage of this method is that the system can be regarded as a functional program, where formal analysis will show, if the system has a unique solution. Also systems that have a stable feedback loop as in Figure 3c are accepted, while the systems of Figure 3a and b are rejected (the result will be the value $\perp$ as solution for the feedback loops). Naturally, the fixed-point approach demands a more sophisticated semantics, but the theory is well understood [49]. Esterel has adopted this approach and the constructive semantics of Esterel is described in [2].

Relation based This approach allows the specification of systems as relations. Thus a system specification may have zero solutions, one solution or multiple solutions. Though an implementation of a system usually demands a unique solution, other solutions may be interesting for high-level specifications. The relation-based approach has been employed in the synchronous language Signal [28].

2.5 Untimed Models

In untimed models there is no global notion of time. If one event does not depend directly or indirectly on another event, it is undefined if one event occurs at the same time as, earlier or later than the other event. Hence, the only ordering on the occurrence of events is determined by causal relationships. If one event depends on another event, it must occur after the other event.

2.5.1 Data Flow Process Networks

Data flow process networks [32] are a special variant of Kahn process networks [26,27]. In a Kahn process network processes communicate with each other via unbounded FIFO channels. Writing to these channels is *non-blocking*, i.e. they always succeed

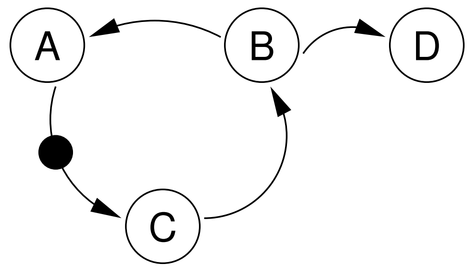


Figure 4: A data flow process network

and do not stall the process, while reading from these channels is *blocking*, i.e. a process that reads from an empty channel will stall and can only continue when the channel contains sufficient data items (*tokens*). Processes in a Kahn process network are *monotonic*, which means that they only need partial information of the input stream to produce partial information of the output stream. Monotonicity allows parallelism, since a process does not need the whole input signal to start the computation of output events. Processes are not allowed to test an input channel for existence of tokens without consuming them. In a Kahn process network there is a total order of events inside a signal. However, there is no order relation between events in different signals. Thus Kahn process networks are only partially ordered which classifies them as an untimed model.

A data flow program is a directed graph consisting of nodes (*actors*) that represent communication and arcs that represent ordered sequences (*streams*) of events (*tokens*) as illustrated in Figure 4. Empty circles denote nodes, arrows denote streams and the filled circles denote tokens. Data flow networks can be hierarchical since a node can represent a data flow graph.

The execution of a data flow process is a sequence of *firings* or *evaluations*. For each firing tokens are consumed and tokens are produced. The number of tokens consumed and produced may vary for each firing and is defined in the *firing rules* of a data flow actor.

Data flow process networks have been shown very valuable in digital signal processing applications. When implementing a data flow process network on a single processor, a sequence of firings, also called a *schedule* has to be found. For general data flow models it is undecidable whether such a schedule exists because it depends on the input data.

Synchronous data flow (SDF) [30, 31] puts further restrictions on the data flow model, since it requires that a process consumes and produces a fixed number of tokens for each firing. With this restriction it can be tested efficiently, if a finite static schedule exists. If one exists it can be effectively computed. Figure 5 shows an SDF process network. The numbers on the arcs show how many tokens are produced and consumed

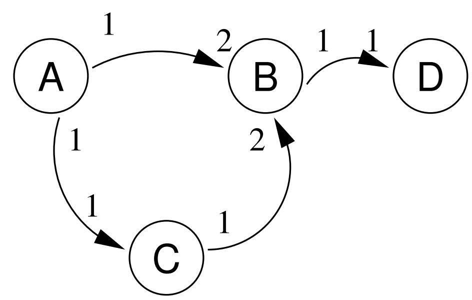


Figure 5: A synchronous data flow process network

during each firing. A possible schedule for the given SDF network is {A,A,C,C,B,D}.

SDF is an excellent example of a MoC that offers useful properties by restricting the expressive power. There exists a variety of different data flow models each representing a different trade-off between interesting formal properties and expressiveness. For an excellent overview see [32].

2.5.2 Rendezvous-based Models

A rendezvous-based model consists of concurrent sequential processes. Processes communicate with each other only at synchronization points. In order to exchange information, processes must have reached this synchronization point, otherwise they have to wait for each other. Each sequential process has its own set of time tags. Only at synchronization points processes share the same tag. Thus there is a partial order of events in this model. The process algebra community uses rendezvous-based models. The CSP (Communicating Sequential Processes) model of Hoare [17] and the CCS (Calculus of Communicating Systems) model of Milner [36, 37] are prominent examples. The language Ada [4] has a communication mechanism based on rendezvous.

2.6 Heterogeneous Models of Computation

A lot of effort has been spent to mix different models of computation. This approach has the advantage, that a suitable model of computation can be used for each part of the system. On the other hand, as the system model is based on several computational models, the semantics of the interaction of fundamentally different models has to be defined, which is no simple task. This even amplifies the validation problem, because the system model is not based on a single semantics. There is little hope that formal verification techniques can help and thus simulation remains the only means of validation. In addition, once a heterogeneous system model is specified, it is very difficult to optimize systems across different models of computation. In summary, while heterogeneous MoCs provide very general, flexible and useful simulation and modeling environment, cross-domain validation and optimization will remain elusive for many years for any heterogeneous modeling approach. In the following an overview of related work on mixed models of computation is given.

In *charts [15] hierarchical finite state machines are embedded within a variety of concurrent models of computations. The idea is to decouple the concurrency model from the hierarchical FSM semantics. An advantage is that modular components, e.g. basic FSMs, can be designed separately and composed into a system with the model of computation that best fits to the application domain. It is also possible to express a state in an FSM by a process network of a specific model of computation. *charts has been used to describe hierarchical FSMs that are composed using data flow, discrete event and synchronous models of computations.

The composite dataflow [22] integrates data and control flow. Vectors and the conversion from scalar values to vectors and vice versa are integral parts of the model. This allows to capture the timing effects of these conversions without resorting to a synchronous or discrete time MoC. Timing of processes is represented only to the level to determine if sufficient data are available to start a computation. In this way the effects of control and timing on dataflow processing are considered at the highest possible abstraction level because they only appear as data dependency problems. The model has been implemented to combine Matlab and SDL into an integrated system specification environment [3].

Internal representations like the system property intervals (SPI) model [50] and FunState [45] have been developed to integrate a heterogeneous system model into one abstract internal representation. The idea of the SPI model is to allow for "global system analysis and system optimization across language boundaries, in order to allow reliable and optimized implementations of heterogeneously specified embedded real-time systems". All synthesis relevant information, such as resource utilization, communication and timing behavior, is extracted from the input languages and transformed into the semantics of the SPI model. An SPI model is a set of parameterized communicating processes, where the parameters are used for the adaptation of different models of computation. SPI allows to model non-determinism through the use of behavioral intervals. There exists a software environment for SPI that is called the SPI workbench and which is developed for the analysis and synthesis of heterogeneous systems.

The FunState representation refines the SPI model by adding the capability of explicitly modeling state information and thus allows the separation of data flow from control flow. The goal of FunState is not to provide a unifying specification, but it focuses only on specific design methods, in particular scheduling and validation. The internal FunState model shall reduce design complexity by representing only the properties of the system model relevant to these design methods.

The most well known heterogeneous modeling and simulation framework is Ptolemy [10, 29]. It allows to integrate a wide range of different MoCs by defining the interac-

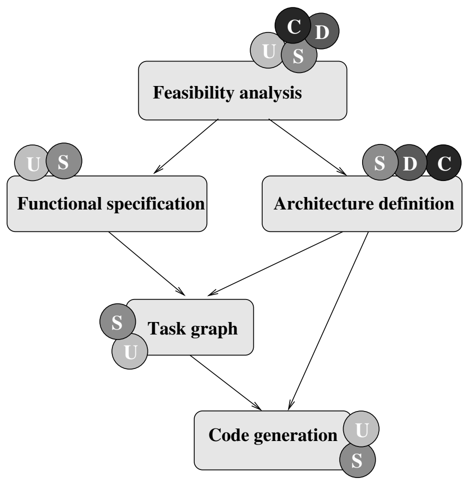


Figure 6: Suitability of MoCs in different design phases. “C”stands for continuous time MoC; “D” for discrete time MoC; “S” for synchronous MoC; and “U” for untimed MoC. More than one label on a design phase means, that all of the MoCs are required since no single MoC is sufficient by itself.

tion rules of different MoC domains.

3 MoCs in the Design Flow

From the previous sections it is evident that different models fundamentally have different strengths and weaknesses. There is no single model that can satisfy all purposes and thus models of computation have to be chosen with care.

Let us revisit the discussed MoCs while considering the different design phases and the design flow. For the sake of simplicity we only identify five main design tasks as illustrated in figure 6. Early on, the feasibility analysis requires detailed studies of critical issues that may concern performance, cost, power or any other functional or non-functional property. The functional specification determines the entire system functionality (at a high abstraction level) and constitutes the reference model for the implementation. Independent of the functional specification is the architecture specification, which may come with performance and functional models of processors, buses and other resources. The task graph breaks the functionality in concurrent activities (tasks), which are mapped onto architecture resources. Once resource binding and scheduling has been performed, the detailed implementation for the resources is created.

The essential difference of the four main computational models that we introduced in the previous section, is the representation of time. This feature alone weighs heavily with respect to their suitability for design tasks and development phases.

3.1 Continuous Time Models

Continuous time MoCs are mostly used to accurately model and analyze existing or prospective devices. They reflect detailed electrical and physical properties with high precision. Hence, they are ideal to study and model tiny entities in great detail but they are unsuitable to analyze and simulate large collections and complex systems due to the overwhelming amount of details. They are usually not used to specify and constrain behavior but may serve as reference models for the implementation. Thus, they are frequently used in feasibility studies, to analyze critical issues, and in architectural models to represent analog or mixed signal components in the architecture. Analog synthesis is still not well automated and hence continuous time models are rarely used as input to synthesis tools.

3.2 Discrete Time Models

The discrete time MoC constitutes a very general basis for modeling and simulation of almost arbitrary systems. With the proper elementary components it can serve to model digital hardware consisting of transistors and gates; systems-on-chip consisting of processors, memories, and buses; networks of computers, clients and servers; air traffic control systems; evolution of prey-predator populations; and much more [5]. In fact it is most popular and widely used in an enormous variety of engineering, economic and scientific applications.

However, it cannot be used for everything. In the context of hardware and software design the discrete time model has the drawback that a precise delay information cannot be synthesized. To provide a precise delay model for a piece of computation may be useful for simulation and may be appropriate for an existing component, but it hopelessly over-specifies the computation for synthesis. Assume a multiplication is defined to take 5ns. Shall the synthesis tool try to get as close to this figure as possible? What deviation is acceptable? Or should it be interpreted as “max 5ns”? Different tools will give different answers to these questions and synthesis for different target technologies will yield very different results and none of them will match the simulation of the discrete time model. The situation becomes even worse, when a delta-delay based model is used. As we discussed in section 2.4 the delta-delay model elegantly solves the problem of non-determinism for simulation, but it requires a mechanism for globally ordering the events. Essentially, a synthesis system had to synthesize a similar mechanism together with the target design, which is an unacceptable overhead.

These problems notwithstanding, synthesis systems for both hardware and software have been developed for languages based on time models. VHDL and Verilog based tools are the most popular and successful examples. They have avoided these problems by ignoring the discrete time model and interpreting the specification according to a clocked synchronous model. Specific coding rules and assumptions allow the tool to identify a clock signal and infer latches or registers separating the combinatorial blocks. The drawbacks of this approach are that one has to follow special coding guidelines for synthesis, that specification and implementation may behave differently, and in general that the semantics of the language is complicated by distinguishing between a simulation and a synthesis semantics. The success of this approach illustrates that mixing different MoCs in the same language is practical. It also demonstrates the suitability of the clocked synchronous model for synthesis but underscores that the discrete time model is not synthesizable.

3.3 Synchronous Models

The synchronous models represent a sensible compromise between untimed and discrete time models. Most of the timing details can be ignored but we can still use an abstract time unit, the evaluation or clock cycle, to reason about the timing behavior. Therefore it has often a natural place as an intermediate model in the design process. Lower level synthesis may start from a synchronous model. Logic and RTL synthesis for hardware design and the compilation of synchronous languages for embedded software are prominent examples. The result of certain synthesis steps may also be represented as a synchronous description such as scheduling and behavioral synthesis.

It is debatable if a synchronous model is an appropriate starting point for higher level synthesis and design activities. It fairly strictly defines that activities occurring in the same evaluation cycle but in independent processes are simultaneous. This imposes an unnecessarily strong coupling between unrelated processes and may restrict early design and synthesis activities too much.

On the other hand in many systems timing properties are an integral part of the system functionality and are therefore an important part of a system specification model. Complex control structures typically require a fine control over the relative timing of events and activities. As single chip systems increas in complexity, this feature becomes more common. Already today there is hardly any SoC design that does not exhibit complex control.

Synchronous models constitute a very good compromise for dealing with time at an abstract level. While they avoid the nasty details of low level timing problems, they allow to represent and analyze timing relations. In essence the clock or evaluation cycle defines *abstract time budgets* for each block. The time budgets turn into timing constraints for the implementation of these blocks. The abstract time budgets constrain the timing behavior without over-constraining it. Potentially there is a high degree of flexibility in this approach if the evaluation cycles of a synchronous MoC are not considered as fixed-duration clock cycles but rather as abstract time budgets, which do not have to be of identical duration in different parts of the design. Their duration could also change from cycle to cycle if required. Re-timing techniques exploit this flexibility. [40, 47].

This feature of offering an intermediate and flexible abstraction level of time makes synchronous MoCs suitable for a wide range of tasks as indicated in figure 6.

3.4 Untimed Models

Untimed models have an excellent track record in modeling, analyzing and designing signal processing systems. They are invaluable in designing digital signal processing algorithms and analyzing their key performance properties such as signal to noise ratio.

Furthermore, they have nice mathematical features, which facilitate certain synthesis tasks. The tedious scheduling problem for software implementations is well understood and efficiently solvable for synchronous data flow graphs. The same can be said for determining the right buffer sizes between processes, which is a necessary and critical task for hardware, software and mixed implementations. How well the individual processes can be compiled to hardware or software depends on the language used to describe them. The data flow process model does not restrict the choice of these languages and is therefore not responsible for their support. For what it is responsible, i.e. the communication between processes and their relative timing, it provides excellent support due to a carefully devised mathematical model.

3.5 Discussion

Figure 6 illustrates this discussion and indicates in which design phases the different MoCs are most suitable. Note, that several MoCs placed on a design phase bubble means that in general a single MoC does not suffice for that phase but several or all of them may be required.

No single MoC serves all purposes equally well. The emphasis is on “equally well” because all of them are sufficiently expressive and versatile to be used in a variety of contexts. However, their different focus makes them more or less suitable for specific tasks. For instance a discrete time, discrete event model can be used to model and simulate almost anything. But it is extremely *inefficient* to use it to simulate and analyze complex systems when detailed timing behavior is irrelevant. This inefficiency concerns both tools and human designers. Simulation of a discrete time model takes orders of magnitude longer than simulation of an untimed model. Formal verification is orders of magnitude more efficient for perfectly synchronous models than for discrete time models. Human designers are significantly more productive in modeling and analyzing a signal processing algorithm in an untimed model than in a synchronous or discrete time model. They are also much more productive to model a complex, distributed system when they have appropriate and high level communication primitives available, than when they have to express all communication with unprotected shared variables and semaphores. Hardware engineers working on the RT level (synchronous

MoC) design many more gates per day than their counterparts not using a synchronous design style. Analog designers are even less productive in terms of designed transistors per day because they deal with the full range of details at the physical and electrical level. Unfortunately, good abstractions at a higher level have not been found yet for analog design with the consequence that analog design is less automated and less efficient than digital design.

MoCs impose different restrictions which, if selected carefully, can lead to significant improvements in design productivity and quality. A strict finite state machine model can never have unbounded memory requirements. This property is inherent in any FSM model and does not have to be proved for every specific design. The amount of memory required can be calculated by static analysis and no simulation is required. This is in contrast to models with dynamic memory allocation where it is in general impossible to prove an upper bound for the memory requirement and long simulations have to be used to obtain a high level of confidence that the memory requirements are indeed feasible. FSM models are restrictive but if a problem suits these restrictions, the gain in design productivity and product quality can be tremendous.

A similar example is synchronous dataflow. If a system can be naturally expressed as an SDF graph, it can be much more efficiently analyzed, scheduled and designed than the same system modeled as a general dataflow graph.

As a general guideline we can state *that the productivity of tools and designers is highest if the least expressive MoC is used that still can naturally be applied to the problem.*

Thus, all the different computational models have their place in the design flow. Moreover, several different MoCs have to be used in the same design model because different sub-systems have different requirements and characteristics. This leads naturally to heterogeneous MoCs which can either be delayed within one language or with several languages under a coordination framework as will be discussed below.

4 Design Activities

Next we investigate specific design tasks and their relation to MoCs. We do not intend to present an exhaustive list of activities, but we hope to illustrate the strong connection and interdependence of design tasks and models on which they operate.

4.1 Synthesis

Today several automatic synthesis steps are part of the standard design flow of ASICs and SoCs. Register Transfer Level (RTL) synthesis, technology mapping, placement and routing, logic and FSM synthesis are among those. Other techniques that have been researched and developed but not successfully commercialized are high level synthesis, system level partitioning, resource allocation and task mapping. We take a closer look at RTL and High-level Synthesis because they are particularly enlightening examples.

4.1.1 RTL Synthesis

RTL Synthesis takes as input an HDL (Hardware Description Language) model of a process, for instance written in VHDL or Verilog, and generates a netlist of gates that adheres to a synchronous design style. Since VHDL and Verilog are simulation not synthesis languages, some of their constructs cannot be synthesized. Every RTL Synthesis tool defines a synthesizable subset of the input language.[4] This subset definition has two objectives. First, constructs that cannot be synthesized into HW are excluded. Obvious examples are file I/O operations and dynamic memory management. Second, typical and efficient HW structures are encoded in the language subset. Synthesis tools will identify FSMs, memories, registers and combinatorial logic in the source model and translate them efficiently onto corresponding HW structures. E.g. VHDL processes have to be written in a specific style with only one clock signal such that the synthesis tool can extract a combinatorial netlist with registers at the outputs. Figure

```
PROCESS (clk, reset)
BEGIN
  IF reset = ´0´ THEN
     state <= 0;
  ELSIF clk'event AND clk = ´1´ THEN
     state <= nextstate;
  END IF;
END PROCESS
```

Figure 7: A VHDL process encoding the P_reg block of figure 8.

7 shows a VHDL process that would be interpreted as a FSM state register by most synthesis tools. If two other combinatorial processes are provided and properly modeled, the tool would derive a FSM structure as shown in figure 8. P_reg reacts to a reset signal to go into the initial state, and to a clock signal to make a state transition.

The definition of a synthesizable subset and the particular interpretation of synthesis tools lead to a divergence of simulation semantics and synthesis semantics. There are three main motivations for this.

1. Some language constructs are pure simulation devices and there is no reason why anybody would want to synthesize them. Examples are file access and assertions.

2. Some language constructs are too expensive to implement in hardware and the current state of the art suggests that they should not be synthesized. Examples

[4] There are different subsets imposed by different tools, but they are not very essential and concern mostly issues of user convenience and synthesis performance rather than the semantics. There exists even an IEEE standard for a synthesizable subset.

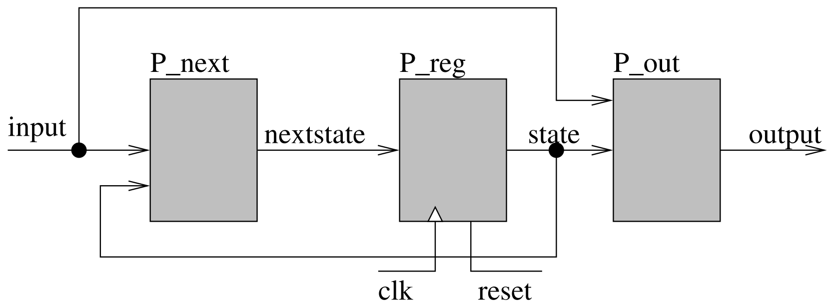


Figure 8: A VHDL synthesis tool derives a state machine when the VHDL description contains three properly modeled processes. P_next is a combinatorial process defining the next state transition function. P_reg is a register storing the state. P_out models the output encoding function.

are multi-dimensional arrays and dynamic memory allocation. When future engineers conclude that such constructs should also be available to hardware designers, these restrictions may disappear.

3. The timing model of the simulation semantics is ill-suited for synthesis. The simulation semantics is based on a discrete time model and allows to express delays in terms of nano and pico seconds. In contrast the synthesized model is a clock synchronous MoC that simply cannot express physical time delays.

The last item interests us most because it shows that VHDL/Verilog based simulation and synthesis use different models of computation, according to our scheme in section 2. The simulation semantics is based on a discrete time MoC which is unsuitable for synthesis. Even if a delay of e.g. 2 ns could be accurately synthesized, it would over-constrain the following technology mapping, placement and routing steps and lead to a hopelessly inefficient implementation. Accurate synthesis of the delta-delay concept is even more elusive.

In contrast, the clocked synchronous MoC[5] allows to separate synthesis of the behavior from timing issues. Since the clock structure and the scheduling of computations in clock cycles is already part of the input model, the RTL synthesis focuses on optimizing the combinatorial blocks between registers. In an analysis step separate from synthesis the critical paths can be identified and the overall system performance can be assessed. Re-timing techniques, that move gates and combinatorial blocks across clock cycle boundaries can shorten critical paths and increase overall performance. If all this proves insufficient the input model to RTL synthesis has to be modified.

In conclusion, for RTL synthesis a clocked synchronous MoC is the best choice because it reflects efficient hardware structures and allows for an effective separation of

[5] Recall from section 2.4 that a clocked synchronous MoC is a synchronous MoC variant where no feed-back loops are allowed within the same clock cycle. Therefore the feed-back loop in figure 8 has to be broken by the P_reg register process.

behavioral synthesis from timing optimization. A lower level, discrete time MoC is entirely inadequate since it over-constrains the synthesis. Starting synthesis with a model based on a higher time abstraction, an untimed MoC, imposes fewer constraints on the synthesis process but consequently requires the synthesis task to include scheduling of operations as will be discussed next.

4.1.2 High-level Synthesis

High-level synthesis, later also called behavioral synthesis, as defined and researched heavily in the 19980s [14], includes the tasks of resource allocation, operation binding and operation scheduling. The input is an algorithm described in a sequential language such as C or as a VHDL process. *Resource allocation* estimates the type and number of HW resources required to implement the algorithm, e.g. how many multipliers, adders, ALUs, etc. *Operation binding* binds operations of the algorithm to allocated resources. *Scheduling* assigns the operations to specific clock steps, thus determining when they will be executed. Figure 9 illustrates the scheduling procedure. From the algorithmic specification in 9(a) the dataflow graph 9(b) is extracted to represent the data dependences. Figure 9(c) shows the scheduled dataflow graph by using the As-Soon-As-Possible (ASAP) scheduling principle.

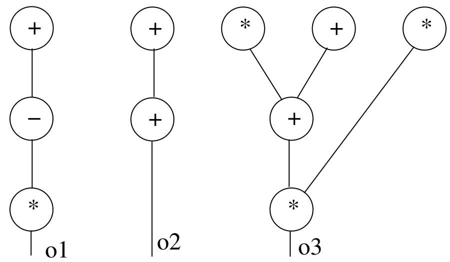
The natural MoC for the input to High-level synthesis is an untimed MoC. Synchronous or discrete time MoCs are unsuitable because they both determine the execution time of individual operations, rendering the scheduling step superfluous. In fact the untimed model was the MoC chosen by all groups that developed high-level synthesis systems. This was either done by defining a dedicated language that could only express an untimed MoC, or by sub-setting a general purpose design language such as VHDL or Verilog. Resource allocation and operation binding concerns the refinement of computation. The abstraction level of the computation and the operators are not defined by the MoCs in section 2. Thus, the untimed MoC is a suitable input to high-level synthesis independent of the kind of operations involved, simple adders and half-adders or highly complex processing elements.

4.1.3 Discussion

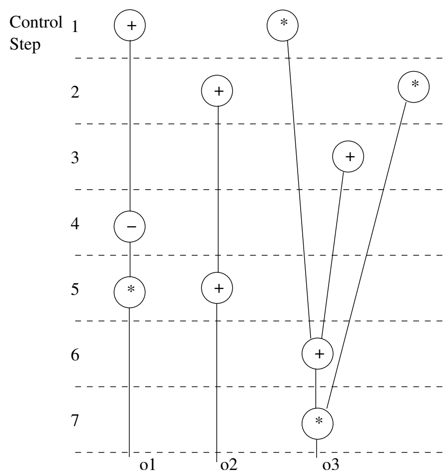
Other synthesis procedures also have their natural input and output MoC. Hence, each synthesis method has to be provided with input models that match its natural MoC, e.g. a clocked synchronous MoC for RTL and an untimed MoC for high-level synthesis. In practice this is accomplished in one of two ways. The obvious approach is to choose an input language that matches well with the natural MoC. If this is not desirable due to other constraints, a language subset and interpretation rules are established, that approximate the MoC required by the synthesis method. We call this technique the *projection* of an MoC into a design language. It is illustrated in figure 10.

```
(1)  a    :=  i1 + i2;
(2)  o1   :=  (a - i3) * 3;
(3)  o2   :=  i4 + i5 + i6;
(4)  d    :=  i7 * i8;
(5)  g    :=  d + i9 + i10;
(6)  o3   :=  i11 * 7 * g;
```

(a) Algorithmic specification



(b) Dataflow Graph



(c) Scheduled Dataflow Graph

Figure 9: An algorithmic specification and its scheduled dataflow graph (from [11]).

Taking a step back we can contemplate the relation between synthesis methods and MoCs. They are mutually dependent and equally important. While it is in general correct that every synthesis method has “natural MoCs” defining its input and output, we can also observe that the major synthesis steps follow naturally from the definition of the MoCs. For every significant difference between two MoCs we can formulate a synthesis step transforming one MoC into the other. On the other hand, the MoCs represent useful abstractions only if we can identify efficient synthesis methods that use them as input and output.

Our treatment of MoCs does not cover other relevant issues such as abstraction and refinement of computation and data types. We have focused foremost on time and therefore we could discuss the scheduling problem of high-level synthesis convincingly while we barely mentioned the allocation and binding tasks. We believe there are good arguments for using time as the primary criteria for categorizing MoCs while other

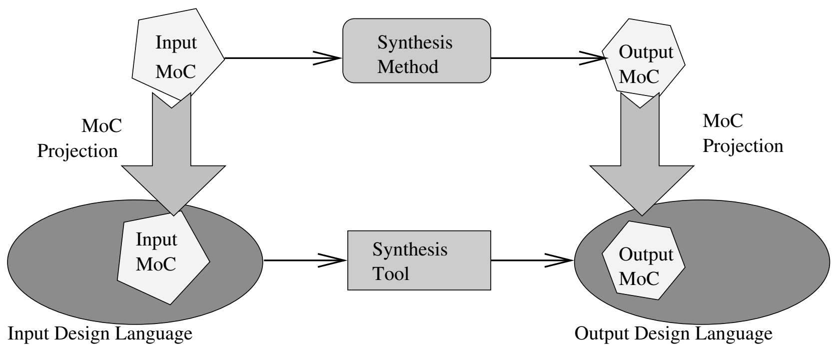


Figure 10: MoC Projection into Design Languages.

domains such as computation, communication and data lead to variants within the same MoC. For a more thorough discussion of this question see [24] or [20]. For a further elaboration of domains and abstractions see [23].

4.2 Simulation

All MoCs that we have discussed can be simulated. So the question that we have to ask is not, which MoC is suitable for simulation, but how efficiently a given MoC can be simulated. Also, we may want to distinguish different purposes of simulation and then we can ask if, for a given purpose, we should prefer one MoC to another.

It is obvious that discrete time MoC simulations are slower than synchronous MoC simulations which in turn are slower than untimed MoC simulations, because MoCs at lower abstraction levels require the computation of many more details. It has been reported that simulations of clock cycle true models, which correspond to our clocked synchronous MoCs, are 1-2 orders of magnitude faster than discrete event simulations, which correspond to our discrete time MoC [41]. Moving to an untimed MoC, e.g. functional or transaction level simulations, can further speed-up simulation by 1 to 2 orders of magnitude [41, 48]. Higher abstraction in any of the domains time, data, computation and communication, improves simulation performance, but the time abstraction seems to be play a dominant role [34], because a higher time abstraction significantly reduces the number of events in a simulation uniformly in all parts of a model.

The disadvantage with abstract MoCs is the loss of accuracy. Detailed timing behavior and the clock cycle period cannot be analyzed in a synchronous MoC simulation. Transaction level models cannot unveil problems in the details of the interface and low level protocol implementation. In an untimed MoC no timing related properties can be investigated and arithmetic overflow effects cannot be observed when using ideal, mathematical data types. Clearly, a trade-off between accuracy and simulation performance, as illustrated in figure 11, demands that a design is simulated at various

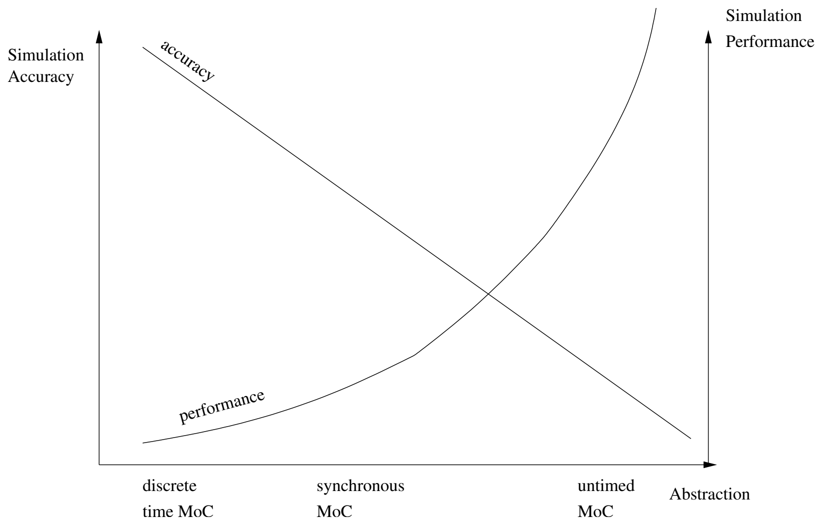


Figure 11: The trade-off between accuracy and simulation performance.

abstractions during a design project from specification to implementation.

4.3 Formal Verification

Formal verification techniques experience a similar trade-off as simulation. If there are too many details in a model, the run-time and memory requirements of most verification tools become prohibitive. Consequently, most techniques are specialized on a particular MoC and sometimes also on a restricted set of properties. They follow the MoCs established by synthesis and design methods, because these have turned out to be useful MoCs for several formal verification techniques as well.

An example formal technique is model checking [7]. It requires a finite state machine (FSM) based model of the design and allows to express and verify various properties such as that a particular variable assignment will never occur in any of the possible states reachable from an initial state. Model checking essentially explores the state space of the FSM until it either finds a counter-example or it can prove the given property, e.g. by exploring the entire reachable state space.

Multiple, communicating FSMs can be handled but only by merging them into a single, flat FSM. This often leads to serious state space explosion problems. Due to clever algorithms and highly efficient data representations can model checking be applied to realistic designs and proves useful in practice.

The natural MoC for property checking is a synchronous MoC, just as for RTL synthesis, since it corresponds to a finite state machine and its evolution. Detailed timing information below the granularity of synchronous MoCs cannot be handled by model checking unless they are encoded in a way fitting into the MoC. On the other

Table 1: Design activities with their respective MoCs (U-MoC = Untimed MoC, S-MoC = Synchronous MOC, D-MoC = Discrete time MoC, C-MoC = Continuous time MoC)

Input MoC	Design task	Output MoC
U-MoC	High-level synthesis	S-MoC
S-MoC	RTL Synthesis	D-MoC
U-MoC	Transaction level simulation	
S-MoC	Cycle-true simulation	
D-MoC	Discrete-event simulation	
C-MoC	Analog simulation	
S-MoC	Model and property checking	

hand an untimed MoC would in principle be compatible with model checking but it would allow for infinitely many ways to merge multiple FSMs into a single one, thus magnifying the state space explosion problem even further.

Just as in the case of synthesis techniques, we can also observe that all formal verification techniques require specific MoCs as input descriptions. The basic principles, such as theorem proving, are often much more general but have to be specialized for a specific problem domain, and thus for a specific MoC, to make them useful in practice. Hence, a MoC serves by dramatically restricting the problem space and, if selected carefully, allows for efficient verification tools.

4.4 Summary

Table 1 summarizes the discussed tasks and gives their respective MoCs.

As mentioned above, we have chosen to distinguish the MoCs according to their time abstraction. Therefore we can naturally analyze design tasks that have a strong relation to a particular time abstraction such as scheduling or cycle-true simulation. For an analysis of all other design tasks in a similar satisfactory way we would have to introduce MoC variants based on computation, data and communication abstractions.

5 Conclusion

We have analyzed the relation between some inherent properties of computational models and various design tasks and phases. Since this is an endeavor far beyond a single article we have taken *time* as our primary parameter and have defined four MoC classes based on the time abstraction: continuous time, discrete time, synchronous time and untimed MoC. This is justified because the chosen representation of time has a critical influence on synchronization, communication and the overall system behavior for systems described by communicating concurrent processes. For a more elaborate study that encompasses all design activities and phases we suggest to still use time abstraction as the primary criterion for defining MoCs but to use other abstractions and domains to introduce more MoC variants as suitable.

We have not carefully illuminated the relation between MoCs and design languages since it is an intricate one with many subtle connections and implications that requires a chapter of its own. For more, but not an exhaustive, elaboration of this issue see [24].

The main targets of our study, complex, heterogeneous, embedded systems, require the use of all presented MoCs. But each MoC has a very specific place and role in the design process as illustrated by figure 6 and table 1. The usage of MoCs should be a conscious choice based on their inherent properties and the given objective and design task. Using them for the wrong purpose will lead to poor results that cannot be rectified by improving a synthesis or simulation algorithm.

But MoCs are not just predefined and given to us and we merely have to pick the right one. Rather, they have to be properly developed and defined for a particular purpose. This is a delicate task because we face a difficult trade-off. To simplify the overall design process and support tool interoperability we would like to have as few different MoCs as possible. However, if we aim at the best possible MoC for a specific task, we will have to integrate many, specialized MoCs in the design flow. History shows, that the process of identifying, accepting and establishing MoCs is tedious and slow. The successful introduction of a new MoC is typically bound to a major paradigm change, such as the move from schematic entry design to RTL based synthesis.

References

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[2] G. Berry. The constructive semantics of pure Esterel - draft version 3. Technical report, INRIA, 06902 Sophia-Antipolis CDX, France, July 2 1999.

[3] P. Bjuréus and A. Jantsch. Modeling of mixed control and dataflow systems in MASCOT. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(5):690–704, October 2001.

[4] G. Booch and D. Bryan. *Software Engineering with Ada*. The Benjamin/Cummings Publishing Company, 1994.

[5] C. G. Cassandras. *Discrete Event Systems*. Aksen Associates, Bosten, Massachusetts, 1993.

[6] C. G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis*. Asken Associates, 1993.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[8] J. Dabney and T. L. Harman. *Mastering SIMULINK 2*. Prentice Hall, 1998.

[9] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127– 144, January 2003.

[10] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[11] P. Eles, K. Kuchcinski, and Z. Peng. *System Synthesis with VHDL*. Kluwer Academic Publisher, 1998.

[12] H. Elmqvist, S. E. Mattsson, and M. Otter. Modelica - the new object-oriented modeling language. In *Proceedings of the 12th European Simulation Multiconference*, June 1998.

[13] M. J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[14] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis*. Kluwer Academic Publishers, 1993.

[15] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[17] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–676, August 1978.

[18] IEEE. *IEEE Standard for Verilog Hardware Description Language*. IEEE, 2001.

[19] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, 2002.

[20] A. Jantsch. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Systems on Silicon. Morgan Kaufmann Publishers, June 2003.

[21] A. Jantsch. Models of embedded computation. In *Embedded Systems*. CRC Press, 2005. Invited contribution; to appear.

[22] A. Jantsch and P. Bjuréus. Composite signal flow: A computational model combining events, sampled streams, and vectors. In *Proceedings of the Design and Test Europe Conference (DATE)*, pages 154–160, Paris, France, March 2000.

[23] A. Jantsch, S. Kumar, and A. Hemani. The Rugby model: A framework for the study of modelling, analysis, and synthesis concepts in electronic systems. In *Proceedings of Design Automation and Test in Europe (DATE)*, 1999.

[24] A. Jantsch and I. Sander. System level specification and design languages. *IEE Proceedings on Computers and Digital Techniques*, 2005. Special issue on Electronic System Design; Invited paper.

[25] S. P. Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

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

[27] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP '77*. North-Holland, 1977.

[28] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Marie. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1335, September 1991.

[29] E. A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley, CA, July 2003.

[30] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, January 1987.

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

[32] E. A. Lee and T. M. Parks. Dataflow process networks. *IEEE Proceedings*, 83(5):773–799, May 1995.

[33] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[34] H. G. M. Khosravipour, G. Gridling. Improving simulation efficiency by hierarchical abstraction transformations. In *Proceedings of the Forum on Design Languages*, 1998.

[35] D. Mathaikutty, H. Patel, and S. Shukla. A functional programming framework of heterogeneous model of computations for system design. In *Proceedings of the Forum on Specification and Design Languages*, Lille, France, September 2004.

[36] R. Milner. A calculus of communicating systems. *LNCS*, 92, 1980.

[37] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[38] H. D. Patel and S. K. Shukla. *SystemC Kernel Extensions for Heterogeneous System Modeling*. Kluwer Academic Publishers, Boston/Dordrecht/London, June 2004.

[39] J. M. Paul and D. E. Thomas. Models of computation for systems-on-chip. In A. Jerraya and W. Wolf, editors, *Multiprocessor Systems-on-Chip*, chapter 15. Morgan Kaufman Publishers, 2004.

[40] F. Rose, C. Leiserson, and J. Saxe. Optimizing synthesis circuitry by retiming. In *Proc. of the Caltech Conference on VLSI*, pages 41 – 67, 1983.

[41] J. A. Rowson. Hardware/software cosimulation. In *Proceedings of the Design Automation Conference*, pages 439–440, 1994.

[42] I. Sander and A. Jantsch. System modeling and transformational design refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[43] J. E. Savage. *Models of Computation, Exploring the Power of Computing*. Addison Wesley, 1998.

[44] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[45] K. Strehl, L. Thiele, M. Gries, D. Ziegenbein, R. Ernst, and J. Teich. FunState - an internal design representation for codesign. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 9(4):524–544, August 2001.

[46] R. G. Taylor. *Models of computation and formal language*. Oxford University Press, New York, 1998.

[47] N. Wehn, J. Biesenack, T. Langmaier, M. Muench, M. Pilsl, S. Rumler, and P. Duzy. Scheduling of behavioural VHDL by retiming techniques. In *Proceedings EuroDAC 94*, pages 546 – 551, September 1994.

[48] B. Werner and P. S. Magnusson. A hybrid simulation approach enabling performance characterization of large software systems. In *Proceedings of MASCOTS*, 1997.

[49] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[50] D. Ziegenbein, K. Richter, R. Ernst, L. Thiele, and J. Teich. SPI—a system model for heterogeneously specified embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(4):379–389, August 2002.