Models of computation and languages for embedded system design

A. Jantsch and I. Sander

Abstract: Models of computation (MoC) are reviewed and organised with respect to the time abstraction they use. Continuous time, discrete time, synchronous and untimed MoCs are distinguished. System level models serve a variety of objectives with partially contradicting requirements. Consequently, it is argued that different MoCs are necessary for the various tasks and phases in the design of an embedded system. Moreover, different MoCs have to be integrated to provide a coherent system modelling and analysis environment. The relation between some popular languages and the reviewed MoCs is discussed to find that a given MoC is offered by many languages and a single language can support multiple MoCs. It is contended that it is of importance for the quality of tools and overall design productivity, which abstraction levels and which primitive operators are provided in a language. However, it is observed that there are various flexible ways to do this, e.g. by way of heterogeneous frameworks, coordination languages and embedding of different MoCs in the same language.

1 Introduction

A system on a chip (SoC) can integrate microcontrollers, digital signal processors (DSPs), memories, custom hardware and reconfigurable hardware, in the form of field programmable gate arrays (FPGAs) together with a communication structure and analogue-to-digital (A/D) and digital-to-analogue (D/A) converters on a single chip (Fig. 1). In total there may be dozens or hundreds of such components on a single SoC. These architectures offer an enormous potential. However, they are also tremendously complex and heterogeneous. This does not only apply for the hardware, but also for the software. Moreover, the overall system complexity grows much 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, synchronisation and parallelism must play a prominent role in all system design languages.

1.1 Hardware and software

In order to manage the complexity and heterogeneity of SoC applications Edwards *et al.* [1] believe that the design approach should be based on the use of one or more formal methods to describe the behaviour of the system at a high level of abstraction, before a decision on its decomposition into hardware and software is taken. The final implementation of the system should be made by using automatic synthesis from this high level of abstraction to ensure implementations that are 'correct by construction'. Validation through simulation or verification should be done at the higher levels of abstraction.

This is in contrast to current design practice, which typically leads to an early definition of the interfaces between hardware and software. After these interfaces have been specified and fixed by system designers, the hardware and software is developed in separate subprojects by different teams. Each of them only validates their design against the specified interfaces, but there is little opportunity to re-evaluate these interfaces or the overall hardware–software partitioning. This is unfortunate because only when more details of the different parts are explicitly modelled, analysed and understood, will certain requirements, constraints and needs become apparent.

Example: In a mixed hardware/software design project a task scheduler shall be implemented in software. During the system specification it has been determined that the hardware timer sets a flag every 50 μs, which is the basic time unit for the scheduler. Even though the used timer has a much higher resolution, during system design and hardware/software interface specification it is decided that the only information passed from the hardware timer to the software scheduler is the flag, which is set by the timer and reset by the scheduler. There is no apparent reason why the interface should be more complicated than this, as previous product generations have successfully used this mechanism. Also, simplicity is a major design goal since experience has shown that over-dimensioned designs have caused increased hardware cost, delayed development and resulted in performance bottlenecks.

It becomes apparent only much later that the new product requires a more sophisticated scheduler. Due to challenging real-time requirements, it sometimes happens that the scheduler is invoked up to 5 μs after the interrupt from the hardware timer. It turns out that it would still be possible for the scheduler to properly schedule all tasks without timing violations, because this situation never happens several times in a row and one specific task can be delayed in that case. For this to work properly the scheduler has to know

IEE Proceedings online no. 20045098

doi: 10.1049/ip-cdt:20045098

Paper first received 28th July and in revised form 3rd November 2004

The authors are with the Department for Microelectronics & Information Technology, Royal Institute of Technology, Electrum 229, Kista 16440, Sweden

E-mail: axel@imit.kth.se

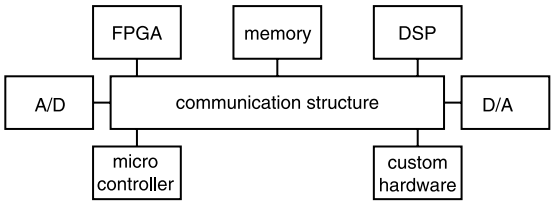

Fig. 1 *Possible system-on-a-chip architecture*

how many μs have elapsed since the last timer tick. Even though the timer has this information readily available, the hardware/software interface has no means to convey it.

More often than not it is very difficult or impossible to change the interface specification. Several design teams work with the given specification, and too much hardware and software had to be changed and revalidated. Thus, if possible, a more local solution to this kind of problem is sought. For instance, in our case the software engineers might be tempted to infer the elapsed time by counting executed instructions. This kind of ad-hoc solution causes a set of problems, such as undocumented dependency on a particular processor type and clock frequency with a nightmare of potential validation problems.

To avoid ending up in ad-hoc problem fixes and local optimisations, Edwards *et al.* [1] argue for a system specification that is detailed enough to identify this kind of problem early on, but which does not commit to specific implementations and detailed interface definitions. Essentially, they want to delay this commitment to a later phase, when the problem and the design are better understood, to make more informed and better decisions. As a consequence the mapping and implementation of the specification must be more automatic, faster and less error-prone.

Thus, they advocate a design process that is based on representations with precise mathematical meaning, so that both the validation and the mapping from the initial description to the various intermediate steps can be carried out with tools of guaranteed performance. A formal model of a design should consist of the following components:

(i) a functional specification given as a set of explicit or implicit relations, which involve inputs, outputs and possibly internal (state) information;
(ii) a set of properties that the design has to satisfy;
(iii) a set of performance indexes that evaluate the design in different aspects (speed, size, reliability, etc.); and
(iv) a set of constraints on performance indexes.

1.2 Computation and communication

A similar view on system design has been formulated by Keutzer *et al.* [2], who also point out that the orthogonalisation of concerns, in particular the separation between (i) computation and communication and (ii) function and architecture is of crucial importance. 'Orthogonalisation of concerns' aims at breaking a complex problem into smaller, simpler pieces. The communication mechanisms between tasks and design blocks should be specified, designed and implemented independently of the design of the computation of the system. When the computations of tasks are designed and optimised, abstract but well defined communication services should be used but not designed. The communication services have to be well defined with respect to functionality and performance. For instance, for a message passing mechanism the synchronisation and buffering policy have to be defined as well as the latency and bandwidth constraints. 'Well specified' is no contradiction to 'high abstraction' and it does not mean there is only one detailed implementation. For instance the delay can be given as a minimum–maximum latency range which is well specified, but still leaves sufficient freedom for various implementation options.

Example: n our case of the scheduler and timer, it is better to specify, design, implement and validate the communication mechanism between hardware and software independent of the design and implementation of the scheduler. The communication defines how data are transfered from the timer to the scheduler, but not which data. Provided there is a well specified but abstract communication mechanism, the system designers can elaborate the system functionality and postpone the details of the hardware/ software interface to a later stage, because the implementation of this interface is straightforward and mostly automatic. It is provided and guaranteed by those that design, implement and specify the communication mechanism.

This example shows that we should replace one type of problem partitioning by another. The old way of partitioning into hardware and software is outdated because it is implementation-oriented and subject to continuous change when implementation options increase. It has to be replaced by a partitioning into computation and communication, which is more problem-oriented, conceptually cleaner and more stable in the face of rapidly developing implementation technology.

1.3 Function and architecture

A similar line of argument can be given for the separation into functionality and architecture. Architectural templates or platforms will be developed and validated independently of the development of the functionality for a particular product. The platform offers services which are both well defined and abstract. Platform implementers can spend great effort in optimising and validating the implementation of these services. Application developers can postpone implementation decisions to a later stage if they only know that the platform fulfills all specified functional and performance requirements. Since platform services can be implemented in a mixture of hardware and software, we see a shift from the traditional hardware/software partitioning to a service provider/service user partitioning, which is more problem-oriented and conceptually cleaner because it is less arbitrary.

1.4 Time

Embedded systems have to fulfill many non-functional requirements. They are usually reactive systems [3] and have to respond continuously to their environment sufficiently fast to meet all timing requirements. Many embedded systems are battery-driven and thus need to be power-efficient. Other embedded systems are safety-critical and must exhibit a minimum level reliability and predictability. How do design languages support these non-functional properties?

With the exception of time, non-functional features have hardly become part of the syntax and semantics of design languages. Time has made it into languages like VHDL and Verilog because, with the introduction of concurrent processes, the relative timing of activities influences the overall system behaviour. Thus, timing becomes an integral part of the functionality. In fact, with concurrency being a major, if not dominant, source of system complexity, the way time is represented and handled in a design language has a considerable impact on the complexity of the design

and validation process. We expect its significance to grow in the coming years. Thus, it is an important theme in this paper and an ordering factor when we discuss models of computation in Section 2.

1.5 Validation

The design process for embedded systems must ensure that the final system implementation complies with the requirements imposed on the system. At present the share of the validation costs compared to the total design costs is continuously increasing. Current estimates suggest that a typical design project employs 1–4 validation engineers for each design engineer. Over the decades there have been many heated discussions about language properties that most effectively support validation. (The term 'validation' is used here for both system simulation and formal verification.) Proponents of C-derived languages such as SystemC [4], Ocapi [5] and others have claimed that the increase of simulation speed, as compared to VHDL and Verilog simulations, facilitates validation. Others [6–8] have argued that simulation, though a very general technique, never suffices and has to be complemented by formal verification. A formal semantics is considered a precondition for a language to be amenable to formal verification. We return to this important issue in Section 4.6.1.

1.6 Abstraction

System design starts with the development of a specification model. In this phase the designer formulates a first model according to the requirements given in a requirement specification, which usually is written in a natural language, e.g. in English. If the specification model is expressed in a formal language, it can be analysed, processed and transformed by tools, which is a significant advantage. (By 'formal language' we mean a language with a formally defined syntax and a well defined semantics. For simplicity we include languages such as C and VHDL even though these languages do not have mathematically defined semantics.) However, a formal design language requires that many details are filled in and decided, which may not be available at this early stage of the project. In order to gain all the obvious benefits the trend has been to introduce formal design languages earlier in the project. The important point, however, is to use a language that offers the 'right abstractions', such that everything that a designer wants to capture can be neatly expressed, while unnecessary details and unavailable information can be ignored.

Example: When a designer has to specify that the brake subsystem of the left front wheel in a car sends status information to the central controller, an abstract communication mechanism for sending data should be used. For example, a `send(data, destination)` primitive only requires to provide the concerned data and the destination of the communication action. This is appropriate since the engineer can focus on what data are sent and where the data are sent to. A tool could verify that the right data are sent to the right place. Alternatively, the designer could use low-level primitives, such as declaring a shared memory location, protecting the shared memory with semaphores, checking the status of the shared memory, writing to it, reading from it and finally acknowledging that the data have indeed been received. This would have several disadvantages. First, these details are not necessary to express the overall system functionality and the specification engineer wastes precious time. Secondly, a verification tool has a much harder task to verify that the correct data are sent, because instead of dealing with one statement it has to process and verify perhaps 20, 50 or 100 statements. Finally, the shared memory mechanism may not be the most efficient mechanism to implement the transaction. Later in the project it may turn out that a message-passing mechanism is more cost- or power-efficient. In that case, the specification engineer's effort was not only wasted, it may also become an obstacle to a more efficient implementation.

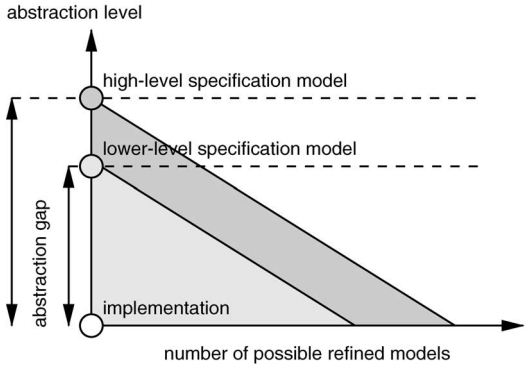

Fig. 2 *High abstraction level leaves a wider design space*

Suppose, the `send` primitive is a non-blocking transaction and does not provide any information to the sender, when the data have been received. Assume further that it is important that the next action by the sender is not initiated before the data are delivered. If there is no blocking-send primitive that synchronises sender and receiver, the specification engineer has to express the desired behaviour in other, perhaps complex, ways resulting in similar disadvantages as described above.

Thus, not only is it important to provide a high abstraction level, it is also crucial to offer the 'right' primitives that are a natural fit to the problem. Moreover, these primitives must also find efficient implementations. As it is very arduous to establish a new abstraction level that meets all these requirements, the march towards higher abstraction levels has been very slow.

The higher the abstraction level of a model, the fewer implementation details are inherent in the model and the larger is the design space. The design space is defined as the number of possible implementations that fulfill a given specification model as illustrated in Fig. 2.

1.7 Refinement

A system specification model serves two distinct purposes [9], as we elaborate further in Section 3. The specification purpose aims at capturing system functionality in an unambiguous, analysable way that allows designers and managers, provider and supplier to discuss and assess the future product. The implementation purpose aims at providing a base and contract for the implementation and validation teams. It must be sufficiently abstract to be developed efficiently and to avoid overconstraining the implementation. It must also have sufficient details to convince everybody that the product can be realised as suggested and respect all requirements.

While a high level of abstraction is imperative for the specification purpose, abstract models are not as suitable for the implementation purpose, as relevant implementation details are missing. Since the specification and the implementation purposes impose incompatible constraints on a model, there is no single model that fits both the specification and the implementation purposes. Thus, the

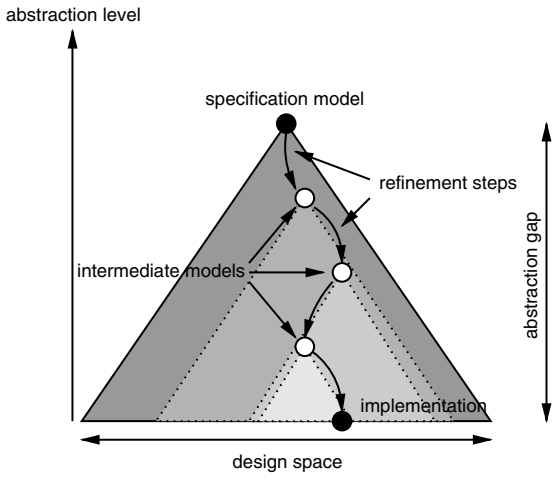

Fig. 3 *Design process is a stepwise refinement from a high-level specification model into a final implementation*

design process has to use several models at different levels as shown in Fig. 3. Starting with a specification model that only includes few implementation details and allows for the application of formal verification techniques, the design process will provide a stepwise refinement technique, which results in an efficient implementation on a complex and heterogeneous architecture. This is not an easy task, since a huge abstraction gap has to be bridged.

Thus, a system design methodology has to offer the following:

- the possibility to model the system at a high level of abstraction allowing for an efficient validation; and
- a refinement methodology that allows the bridging of the abstraction gap in order to yield an efficient implementation.

These objectives can be summarised as the challenge for a successful system design methodology. From Fig. 3 we can clearly see that models are used at several levels of abstraction. Since the specification and implementation purposes require different models, it is not very likely that the same kind of model can be used for both purposes. Thus, the design process has not only to provide a refinement of a particular model of computation, but also mapping rules between different models of computation.

In the following Section we discuss models of computation (MoC). We view MoC as a convenient concept that abstracts slightly from the languages and allows us to focus on the essential issues of concurrency, time, communication and synchronisation. The purpose of a specification model is discussed in more detail, which determines the requirements for the specification language. This discussion allows us to appreciate the strengths and weaknesses of different MoCs with respect to the requirements of the design process. Languages are discussed. However, we do not give a complete list of design languages, but we only focus on features concerning time and communication.

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 [10–14]), we use it to define the time representation and the semantics of communication and synchronisation 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 [15, 16]. 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 [17], the functional language Haskell [18] is used to express several models of computation. Libraries have been created for synchronous, data flow and discrete event models of computation. Similarly, standard ML has been used to implement timed, synchronous and untimed computational models [19]. SystemC has also been extended to support SDF (synchronous dataflow) and CSP (communicating sequential processes) models of computation in addition to its native discrete event MoC [20].

To choose the right model of computation is of utmost importance, since each MoC has certain properties. As an example, consider a process network modelled as a discrete event 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 [20] 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 [21]. Their community faces similar problems in the design of embedded systems. In fact all typical parallel computer structures (SIMD, MIMD) can be implemented on a SoC architecture. (Flynn has classified typical parallel data structures in [22], where SIMD is an abbreviation for single instruction, multiple data and MIMD for multiple instruction, multiple data.) Recognising 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 emphasise that a model should hide most of the details (decomposition, mapping, communication, synchronisation) from programmers if they are to 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 does not even have 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.

Depending on what information is explicit in a model they distinguish six levels, i.e.

(1) nothing explicit
(2) parallelism explicit
(3) parallelism and decomposition explicit
(4) parallelism, decomposition and mapping explicit
(5) parallelism, decomposition, mapping and communication explicit
(6) parallelism, decomposition, mapping, communication and synchronisation explicit

Next, we present a number of important models of computations and give their key properties. Following [10, 16], we organise them according to their timing

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 [11]. There each event has a time tag and different time tag structures result in different MoCs. For example, 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 organised along other criteria, e.g. along the kinds of elements manipulated in a MoC which led Paul and Thomas [12] to a grouping of MoCs for hardware artefacts, MoCs for software artefacts and MoCs for design artefacts. However, an organisation along properties that are not inherent properties of MoCs is of limited use because it changes when MoCs are used in different ways.

A drawback of an organisation along the time abstraction is that all strictly sequential models, such as finite state machines and sequential algorithms, all fall into the same class of MoCs, where the representation of time is irrelevant. However, this is of minor concern to us, since we focus on parallel MoCs.

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 [23], VHDL–AMS and Modelica [24]. The behaviour is typically expressed as equations over real numbers. Simulators for continuous-time MoCs are based on differential equation solvers that compute the behaviour of a model, including arbitrary internal feedback 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 modelled with continuous time such as analogue and mixed-signal components.

To be able to model and analyse a complete system that contains analogue components, mixed-signal languages and simulators such as VHDL–AMS have been developed. They allow modelling of the pure digital parts in a discrete-time MoC and the analogue 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 a clear benefit.

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 integer or natural numbers, are called discrete-time models. (Sometimes this group of MoCs is denoted as 'discrete-event MoC'. Strictly speaking 'discrete event' and 'discrete time' are independent, orthogonal concepts. In discrete-event models, the values of events are drawn from a discrete set. 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 [25] for a good coverage of discrete event models.)

Discrete-time models are often used for the simulation of hardware. Both VHDL [26] and Verilog [27] use a discrete-time model for the their simulation semantics. (Both languages have a different model of computation for synthesis, which is similar to a perfect synchronous model due to the use of synchronous registers with the difference that computation does not have a zero delay). A simulator for discrete-time MoCs is usually implemented with a global event queue that automatically 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, such as integers, but the elementary time unit is not a physical unit but more abstract due to two abstraction mechanisms:

(i) The timing of activities and events is not precisely defined but only constrained by the beginning and end of the elementary time slot.
(ii) The timing of intermediate events that are not visible at the end of an elementary time slot is irrelevant and can be ignored.

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

The synchronous assumption can be formulated according to [3]. 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 modelling 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 behaviour 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 behaviour 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 timed and synchronous models

Timed models allow zero-delay computation; in synchronous models this is even a basic assumption. As a consequence, feedback loops may introduce inconsistent behaviour. In fact, feedback loops as illustrated in Fig. 4 may have no consistent solution; there may be one consistent and unambiguous solution or there may be many solutions.

Figure 4*a* 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 4*b* shows a system with a feedback loop with multiple solutions. Here the system is stable, if both AND

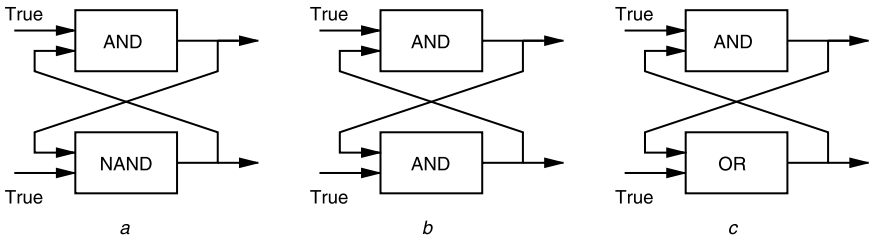

Fig. 4 *Feedback loop in a synchronous system*
a No solutions
b Multiple solutions
c Single solution

functions have `False` or if both AND functions have `True` as their output value. Thus, the system has two possible solutions.

Figure 4*c* shows a system with 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 Fig. 4*a* 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 nondeterminism. Nondeterminism 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 timed and synchronous models are of such importance there are several approaches which address this problem [15].

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. In order to solve the zero-delay feedback problem, 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 infinitesimally small amount of time. Each operation takes zero time units, but 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 Fig. 4*a* 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 Fig. 4*b* has two solutions, since the simulation semantics of VHDL would assign an initial value for the output of the AND gates (`False`) and thus would only give one stable solution, concealing the nondeterminism from the designer. (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.) Another serious drawback of the microstep concept is that it leads to a more complicated semantics, which aggravates the task of formal reasoning.

Forbid zero-delays: The easiest way to cope with the zero-delay feedback problem is to forbid them. In cases of Fig. 4*a* and Fig. 4*b* this would mean the insertion of an extra delay function, e.g. after the upper AND function. Since a delay function has an initial value, the systems will stabilise. Assuming an initial value of `True`, Fig. 4*a* will stabilise 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 4*b* would stabilise with the output value `True` for both AND functions. A possible problem with this approach is that a stable system, such as Fig. 4*c*, is rejected, since it contains a zero-delay feedback-loop. This approach is adopted in the synchronous language Lustre [28].

Unique fixed-point: The idea of this approach is that a system is seen as a set of equations for which one solution in the 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 Fig. 4*c* are accepted, while the systems of Fig. 4*a* and Fig. 4*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 [29]. Esterel has adopted this approach and the constructive semantics of Esterel is described in [30].

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 [31].

2.5 Untimed models

2.5.1 Data flow process networks:

Data flow process networks [32] are a special variant of Kahn process networks [33, 34] 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 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.

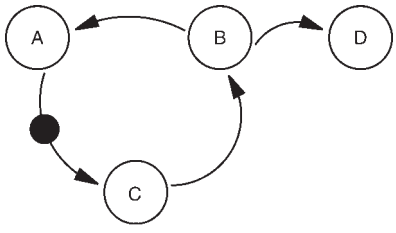

Fig. 5 *Data flow process network*

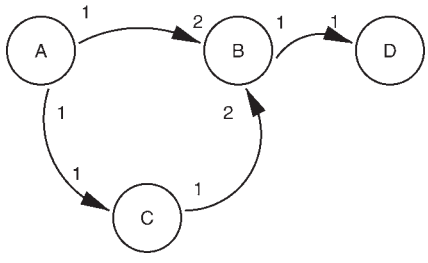

Fig. 6 *Synchronous data flow process network*

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 Fig. 5. Empty circles represent nodes, arrows represent streams and the filled circle represents a token. 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 to be 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 cannot be decided whether such a schedule exists because it depends on the input data.

Synchronous data flow (SDF) [35, 36] 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 6 shows an SDF process network. The numbers on the arcs show how many tokens are produced and consumed during each firing. A possible schedule for the given SDF network is {A,A,C,C,B,D}.

There exists a variety of different data flow models. 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 synchronisation points. In order to exchange information, processes must have reached this synchronisation point, otherwise they have to wait for each other. In the tagged signal model each sequential process has its own set of tags. Only at synchronisation points do 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 [37] and the CCS (calculus of communicating systems) model of Milner [38, 39] are prominent examples. The language Ada [40] 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 optimise systems across different models of computation. In summary, while heterogeneous MoCs provide very general, flexible and useful simulation and modelling environment, cross-domain validation and optimisation will remain elusive for many years for any heterogeneous modelling approach. In the following an overview of related work on mixed models of computation is given.

In $*$charts [41], 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 [42] 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 capture of the timing effects of these conversions without resorting to a synchronous or timed 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 [43].

Internal representations sucha as the system property intervals (SPI) model [44] 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 optimisation across language boundaries, in order to allow reliable and optimised implementations of heterogeneously specified embedded real-time systems'. All synthesis relevant information, such as resource utilisation, communication and timing behaviour, is extracted from the input languages and transformed into the semantics of the SPI model. An SPI model is a set of parameterised communicating processes, where the parameters are used for the adaptation of different models of computation. SPI allows modelling of non-determinism through the use of behavioural 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 modelling 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 reduces design complexity by representing only the properties of the system model relevant to these design methods.

The best known heterogeneous modelling framework is Ptolemy. It allows integration of a wide range of different MoCs by defining the interaction rules of different MoC domains. We come back to Ptolemy in Sections 4.3 and 4.5.

3 Purpose of models

From the previous Sections it is evident that different models fundamentally have different strengths and weakness. There is no single model that can satisfy all purposes and thus models of computation have to be chosen with care. If we again consider the design flow, we can distinguish two purposes of a system model [9]:

Specification purpose: The model is used to develop the system functionality and to study if it is indeed a solution to the imposed problem and all requirements can be fulfilled by the model. The specification model abstracts from implementation details and allows for a large design space, where many parts of the hardware and software architecture are not yet determined.

Implementation purpose: The model will be efficiently mapped onto the given architecture. Here, implementation details play a very important role and also the underlying architecture has to be reflected by the model. On the other hand, the specification should not unnecessarily restrict the implementation by prescribing details that cannot be efficiently implemented. This is a very fine balance because too abstract a specification model will make synthesis tools inefficient and infeasible. A too detailed model will over-constrain the design process and lead either to abandoning the specification model or to an inefficient and overly costly implementation; most likely both. Even worse, this balance is not only hard to strike, it also changes with advances of implementation technology, design technology and is different in different application areas.

Let us revisit the discussed MoCs in the light of this observation and with respect to the design flow. For the sake of simplicity we only identify six main design tasks as illustrated in Fig. 7. Early on in the requirements definition phase a MoC needs to be able to efficiently capture the main functional features without bothering about details. In addition, feasibility analysis requires detailed studies on 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. 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.

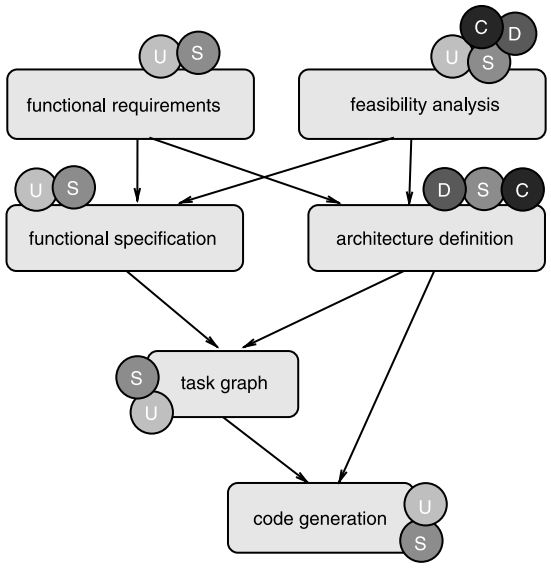

Fig. 7 *Suitability of MoCs in different design phases*
'C' stands for continuous-time MoC, 'D' for discrete-time MoC, 'S' for synchronous MoC and 'U' for untimed MoC

The essential difference of the three main computation 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 analyse existing or prospective devices. They are usually not used to specify and constrain behaviour but may serve as reference models for the implementation. Thus, they are frequently used in feasibility studies, to analyse critical issues, and in architectural models to represent analogue or mixed-signal components in the architecture. Analogue synthesis is still in its infancy and hence continuous-time models are rarely used as input to synthesis tools.

3.2 Discrete-timed models

The discrete-timed model has the drawback that precise delay information cannot be synthesised. The ability 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 5 ns. Should the synthesis tool try to get as close to this figure as possible? What deviation is acceptable? Or should it be interpreted as 'max 5 ns'? Different tools will give different answers to these questions and synthesis for 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 synthesise 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 timed 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 simulation and 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 synthesisable.

3.3 Synchronous models

Synchronous models represent a sensible compromise between untimed and fully timed 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 behaviour. 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 behavioural synthesis.

It is debatable whether 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 a rather 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 become more complex, this feature becomes more common. Already today there is hardly any SoC design that does not exhibit fairly complex control algorithms.

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 representation and analysis of 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 behaviour 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. Retiming techniques exploit this flexibility [46, 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 Fig. 7.

3.4 Untimed models

Untimed models and their variants 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 7 summarises 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 fully timed, discrete-event model can be used to model and simulate almost anything. But it is extremely inefficient to use it to simulate and analyse complex systems when detailed timing behaviour is irrelevant. This inefficiency concerns both tools and human designers. Simulation of a timed MoC model takes orders of magnitude longer than simulation of an untimed MoC model. Formal verification is orders of magnitude more efficient for perfectly synchronous MoC models than for timed MoC models. Human designers are significantly more productive in modelling and analysing a signal processing algorithm in an untimed MoC model than in a synchronous or timed MoC model. They are also much more productive in modelling 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. Analogue designers are even less productive 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 analogue design, with the consequence that analogue 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 analysed, scheduled and designed than the same system modelled 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 subsystems 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 Relation of design languages and MoCs

Let us revisit the models of computation again to see how they appear in different, popular design languages

4.1 General-purpose design languages

We call design languages such as VHDL, Verilog or SystemC 'general-purpose design languages' because they constitute the backbone of most ASIC, FPGA, SoC and HW-SW codesign flows. They can do this because they are based on a very general discrete-time MoC that allows us to model and simulate most of the parts of a typical SoC. They also have a wider range of powerful other features that allow for advanced data structuring and modelling, for building advanced functional hierarchies with functions, procedures and processes, and for managing large and complex design projects. However, since a discrete-time MoC has severe restrictions as discussed above, these languages have been extended into several main directions.

VHDL is a case in point. Facing the need to model and simulate entire systems including digital and analogue parts, VHDL has been extended to VHDL–AMS [48]. The idea is to combine two distinct MoCs and two different simulation engines under a unifying syntax. Thus, VHDL–AMS in fact implements a heterogeneous MoC consisting of a continuous-time and a discrete-time MoC.

Originally VHDL was devised as a general purpose simulation and modelling language for digital hardware. But it became soon apparent that it is highly desirable to use the same language as input to synthesis tools. For reasons discussed above, a discrete-time model is not suitable as an input to a hardware synthesis tool. The solution was to define how a synchronous MoC is expressed in VHDL syntax. Thus, the so called 'synthesisable subset' of VHDL essentially defines a clocked synchronous MoC, which is well suited as an input to synthesis. For similar reasons formal verification tools are also operating on the synchronous MoC expressed in VHDL and not on the discrete-time MoC of the VHDL simulation semantics.

SystemC is taking a similar evolutionary route. Based on a general discrete-time MoC, a synthesisable subset has been established [49, 50] that essentially defines a clocked synchronous MoC. There are also recent attempts to extend SystemC to analogue modelling [51]. But SystemC takes the idea to support several MoCs further by explicitly defining a transaction level of abstraction [4]. From a time perspective the transaction level of abstraction is an untimed MoC because, if strictly used without other timing constructs interfering, it imposes only a partial order on the timing of events solely determined by the sequence of interaction between processes. However, since it is not strictly enforced and can freely be mixed with other MoCs, synthesis, analysis and verification tools can hardly make use of it. For instance, even if all processes in a process network would consume and produce a constant number of data items during each evaluation and the process network would in fact constitute a synchronous data flow (SDF) process network, a tool would most likely not be able to derive a static schedule because it could not prove that the network is an SDF graph.

Therefore, Patel and Shukla have proposed an alternative way to equip SystemC with other MoCs [20]. They provide different simulation kernels for other MoCs, such as SDF and communication sequential processes, in addition to the discrete-event simulation kernel. This has the consequence that the designer explicitly flags where which MoC is used, and a tool can safely make assumptions based on specific properties of a MoC.

In summary, the trend for these general purpose design languages is to define subsets and modelling rules to realise different MoCs for various purposes. We call this technique 'embedding an MoC' and we expect this to become one of the main roads to advance design technology. It has proved very successful in the past even though it has often been applied ad hoc.

4.2 Synchronous languages

The synchronous languages [3, 52, 53] have been successfully used in the area of reactive and safety-critical embedded control systems. These languages are based on the perfectly synchronous computational model (Section 2.3). This model gives a solid mathematical foundation for formal reasoning and the application of formal program manipulation techniques. The synchronous languages have the following key properties:

- they support concurrency;
- they have a simple and elegant formal semantics that allows parallel composition to be expressed in a clean way;
- they support the concept of synchrony, which divides time into discrete instants.

The following part focuses largely on the presentation of the imperative synchronous language Esterel. Then follows a short discussion about other synchronous language.

Esterel: Esterel [54–56] is an imperative language that is very suitable for the description of control. A program consists of a collection of nested, concurrently running threads that are described in an imperative syntax. Threads communicate with each other by means of signals. In addition to common control structures such as `if-then-else`, Esterel has a large number of pre-emption statements that allow the termination of statements. There is a formal framework developed for Esterel, which includes causality analysis ensuring that causality constraints are never contradictory in any reachable state.

Consider the program `M1` in Fig. 8 [57]. It outputs an event `O` as soon after both input events `A` and `B` have been received. In addition, whenever input event `R` is received, the module is reset.

Lines (2) and (3) define the input and output events of the module. The loop in lines (4) to (6) loops forever because it has no exit condition. In line (5) the module waits for events `A` and `B` in parallel. The operator '| |' is a parallel operator, while ';' is a sequence operator. The brackets '[' and ']' structure the text. Thus, in line (4) the module waits for both events in parallel and synchronises as soon both events have been received. Then, event `O` is emitted. The sequence operator does not take any time; thus `O` is in fact emitted at

```
(1)  module M1:
(2)  input A, B, R;
(3)  output O;
(4)  loop
(5)     [ await A || await B ];
(6)     emit O
(7)  each R
(8)  end module
```

Fig. 8 *Simple Esterel module with three inputs and one output*

the same time instant as the last of the two events A and in a B is received. After O has been emitted the next iteration of the loop is started and the module waits for the next inputs.

When event R is received, the module is immediately aborted and restarted. R works as a reset signal interrupting the loop or any other activity in the module.

Esterel has been designed for synchronous/reactive systems, where a program typically waits for inputs, computes something and emits outputs. Modules communicate with each other via events. An event emitted by one module, such as O by module M1, is instantaneously seen by all other modules. This is called 'instantaneous broadcast'. Thus a module does not need to know the name or address of a receiving module to communicate with it.

Esterel follows the perfect synchrony assumption that neither computation nor communication takes any observable time. All activities of a module or system are synchronised with incoming events. A module like M1 waits for inputs and reacts to them instantaneously.

As a consequence, zero-delay feedback loops become possible. Consider the Esterel module in Fig. 9. In line (3) the module checks with present O1 if the event O1 is occurring. If so it emits O2. In parallel (line (5)) it checks for the presence of O2 to emit O1. The program is nondeterministic because it describes two possible, consistent behaviours. Both events O1 and O2 can be present or both can be absent. Hence, M2 is not a legal Esterel program.

These dependency cycles can be broken by prohibiting zero-delay loops. For instance, Fig. 10 shows how M2 can be corrected. The operator pre refers to the previous value of a signal. Thus, in line (3) it is checked if the event O1 has occurred in the previous evaluation cycle. M3 would work as follows: if O1 has occurred in the previous cycle, O2 would be emitted in the current cycle. As a consequence, O1 would also be emitted in the current cycle due to lines (6–7) which check the presence of O2 in the current cycle. M3 has no zero-delay feedback loop and is a legal Esterel program.

However, there exist also legal Esterel programs with zero-delay feedback loops. Consider module M4 in Fig. 11. Even though we have a cyclic dependency of O1 and O2 and *vice versa*, the two halves of the cycle can never occur simultaneously. O2 only depends on O1 when event I is present, while O1 only depends on O2 when I is not present. Thus, M4 has a well defined behaviour and is a correct and legal Esterel program. The constructive semantics of Esterel

```
(1)  module M2:
(2)  output O1, O2;
(3)     present O1 then emit O2 end
(4)  ||
(5)     present O2 then emit O1 end
(6)  end module
```

Fig. 9 *Illegal Esterel module due to zero-delay feedback*

```
(1)  module M3:
(2)  output O1, O2;
(3)     present pre(O1)
(4)        then emit O2 end
(5)  ||
(6)     present O2
(7)        then emit O1 end
(8)  end module
```

Fig. 10 *Dependency cycle of M2 has been broken making M3 correct*

```
(1)  module M4:
(2)  input I;
(3)  output O1, O2;
(4)  present I then
(5)     present O1 then emit O2 end
(6)  else
(7)     present O2 then emit O1 end
(8)  end module
```

Fig. 11 *Legal Esterel module with a zero-delay feedback loop*

[30] defines the set of legal programs and provides also a constructive way to efficiently distinguish legal from illegal programs.

Other synchronous languages: Esterel has been developed for control dominated synchronous/reactive systems. Other synchronous languages for control dominated systems are Statecharts [58] and Argos [59]. Both have a graphical syntax for describing state machines.

Synchronous languages for dataflow dominated systems are Lustre and Signal.

Lustre [28] is a declarative data flow language where systems are composed of sets of equations. Each variable is a function of time and denotes a flow. Operators operate not on single values, but on whole flows. We will discuss in more detail the ideas of dataflow languages in the context of untimed MoCs in Section 4.3, where we introduce the language Lucid.

Similarly, Signal is also a declarative dataflow language, but it is based on relations rather than functions. A Signal [31] program is a set of constraints or relations on the involved signals. The Signal compiler performs formal calculations on synchronisation, logic and data dependencies to check program correctness and produce executable code.

Both Lustre and Signal support multiple clocks and clock domains.

The synchronous languages have been successfully used in industry and there exist industrial tools for Esterel (from Esterel Technology), Lustre (Scade) and Signal (Sildex). Recently Esterel Technology has acquired the Scade environment in order to be able to combine a control-oriented (Esterel) with a dataflow-oriented (Lustre) synchronous approach.

Synchronous language programs are usually translated to finite-state automata in order to implement them as a sequential reactive program on a single processor. However, Esterel [60] and Lustre [61] have been also translated into hardware implementations.

The clean mathematical formalism has led to the development of several verification tools for the synchronous languages. Halbwachs and Raymond [62] give an overview over the techniques and tools developed for the validation of reactive systems described in Lustre. However, these techniques can be adapted to any synchronous language.

4.3 Dataflow languages

Dataflow languages can be traced back to the 1970 s when Dennis [63], Kahn [33], Ashcroft [64] and others pioneered this field. Although Kahn has not explicitly targeted dataflow machines with his programming language [33], it laid the foundation for dataflow process networks and its many derivations such as synchronous dataflow [36], cyclostatic dataflow [65], Boolean-controlled dataflow

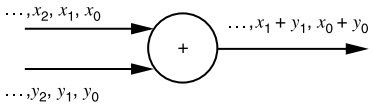

Fig. 12 *Dataflow node adding pointwise the values of the input streams*

[66] and many others. Also, Kahn process networks coincide strongly with the untimed MoC of Section 2.5.1.

We introduce Lucid to illustrate some of the principles of dataflow languages. Then we mention briefly a few other languages and graphical notations.

Lucid: Dataflow languages have operators, often called 'nodes', that operate on streams of input values to generate streams of output values. Assuming, that X denotes the stream of values $\langle x_0, x_1, x_2, \ldots \rangle$ and Y denotes $\langle y_0, y_1, y_2, \ldots \rangle$, then $X + Y$ would denote the pointwise addition of the two streams and result in the stream $\langle x_0 + y_0, x_1 + y_1, x_2 + y_2, \ldots \rangle$ as illustrated in Fig. 12.

In Lucid [64] every data object, be it a variable or constant, is a stream, i.e. a potentially infinite sequence of values. Hence, $X + Y$ from above would be a valid Lucid expression operating over the stream variables X and Y. The constant '3' would denote an infinite sequence where each element is the number 3. Thus, the expression $X + 3$ denotes the sequence $\langle x_0 + 3, x_1 + 3, x_2 + 3, \ldots \rangle$.

In addition to pointwise operators, Lucid has also special non-pointwise operators. For instance, `first` X results in a constant stream with the first element of X. `next` X drops the first element, e.g. `next` $X = \langle x_1, x_2, x_3, \ldots \rangle$.

Consider the following Lucid program [64]:

```
(1) N = first input
(2) first I = 0
(3) first J = 1
(4) next J = J + (2 × I) + 3
(5) next I = I + 1
(6) output = I as soon as J>N
```

All variables are streams and the equations above define the streams. Line (1) defines the constant stream N that is determined by the first input value. Lines (2) and (5) define stream I, which represents the natural numbers starting from 0. Lines (3) and (4), the core of this program, define stream J that depends on I and recursively on itself. It begins $J = \langle 1, 4, 9, 16, \ldots \rangle$, i.e. it represents the squares of the natural numbers. Finally, line (6) defines the output, which is a constant stream taken from one value of I. It is the least I for which $(I + 1)^2 > N$. Thus, the program computes $\lfloor \sqrt{N} \rfloor$.

An alternative interpretation of this Lucid program is that of a loop with I being the iterator and line (6) constitutes the exit condition.

Lucid has later developed into a multidimensional dataflow language [67, 68] and into Granular Lucid (GLU) [69], a coordination language for coarse-grain parallelism, where sequential parts are described in C. In several aspects it is typical for dataflow languages. Many dataflow languages are based on a functional and declarative semantics. They have originally mostly been used for regular, computation intensive applications, such as matrix manipulations, graphics manipulations and many other scientific applications. Many of them have also been used for programming parallel computers because the functional paradigm facilities parallelisation and the problems in scientific computing are so demanding that researchers have always sought to exploit parallel architectures.

Other dataflow languages: With the rise of signal processing in the telecommunication domain several companies have commercialised tools based on the principles of dataflow computation. Most of these tools are based on graphic block languages, such as SPW from Cadence/CoWare and LabView from National Instruments.

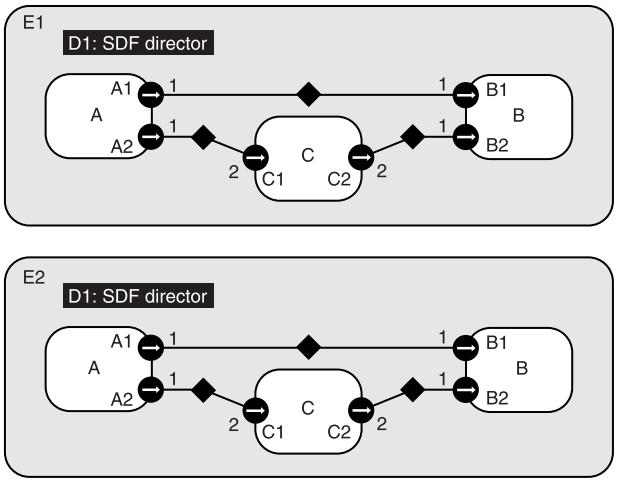

Fig. 13 *Ptolemy SDF example [70]*

As an example of a graphical block language based on an untimed, dataflow MoC, consider Fig. 13, which is an SDF graph in Ptolemy II [70]. `A`, `B` and `C` are processes. They attach via ports to channels and the arrows in the ports determine the direction of the dataflow in the channels. The numbers close to the ports denote the number of data tokens consumed and produced during each invocation of a process. `E1` shows a valid SDF graph, for which a static schedule can be found, while `E2` is not a valid SDF graph, since for any execution sequence either the processes would block or one of the channel buffers will grow beyond any bound.

It is important to note, that these graphical block languages can be considered as coordination languages because they do not define how the processes are modelled. In principle the processes can be described in any language as long as they obey the rules of the governing MoC when interacting with the environment.

4.4 Rendezvous-based communication

Many languages, partially influenced by Hoare's communicating sequential processes (CSP) [37] and Milner's calculus of communication systems (CCS) [38], have defined a communication mechanism with tight synchronisation between the partners, which is often called a rendezvous. In a rendezvous, both sender and receiver block until the other partner is ready and the act of communication has completed. This means that every communication leads to a synchronisation point leading to a tighter coupling between the processes than would be necessary due to data dependencies. Furthermore, it means that channel buffers are not required. In the dataflow models discussed above, only the receiver blocks if no input data is available, but the sender just sends the data without concern for the readiness of the receiver. This means that a data item has to be buffered in the channel during the time after it has been sent and before it has been consumed. To determine the necessary buffer size and to avoid buffer overflow is one of the main design challenges in implementing dataflow models. This is not a problem for rendezvous-based communication because all channels need to buffer at most one datum.

Handel-C: Handel-C [71] is a C variant targeting complex FPGA design. It differs from ANSI C in a number of ways, most importantly by offering fine control over parallelism and timing. Arguing that for FPGA design a discrete-time

```
// Process A: primary clock domain
set clock = external "R25";
unsigned 4 result;

//channel defined in other file:
extern chan unsigned 4 CHANNEL;
void main(void)
{ while(1)
  { delay;
    CHANNEL ? result;
    //process will wait
    //until data received
  }
}
```

```
// Process B: secondary clock domain
set clock = external_divide "R25" 2;
unsigned 4 x;

chan unsigned 4 CHANNEL;

void main(void)
{ while(1)
  { x++;
    CHANNEL ! x;
    //process will wait
    //until data received
  }
}
```

Fig. 14 *Process B sends 4-bit data to process A in Handel-C*

model is superfluous and ineffective, it offers only two higher-level MoCs. Each module, as defined by a `main()` function, is governed by one clock according to the clock synchronous MoC. Within a module, `par` and `seq` statements allow for tight control over parallelism. Several `main()` functions can be declared and each is governed by a different clock. Essentially they constitute a process network of a rendezvous-based untimed MoC. It is untimed because the Handel-C model does not determine the relation between the different clock signals, which is left to the synthesis and implementation. Communication between `main()` processes is strongly coupled because both sender and receiver block until the communication is completed.

Consider the example in Fig. 14. Processes A and B are defined in two separate files with two different `main()` functions. The `set clock` lines define the clocks for the two processes. In this case the process B clock is derived from process A clock, running at half the speed. This is not necessary and both processes could use entirely independent clocks. Note also, that information about relative frequencies of clocks is not used for determining the behaviour of the system; hence it is a truly untimed MoC.

The `chan` keyword defines the communication channel, which can only be one-directional. The direction is determined by its first usage. The token '`!`' denotes the writing to a channel, while '`?`' denotes reading from a channel. Both processes are blocked until the communication is completed. Thus the two loops are progressing in a lock step manner even though process A runs on a clock twice as fast as process B.

4.5 Heterogeneous frameworks

Due to the complexity and heterogeneity of embedded systems and SoCs framework and languages that support multiple MoCs have become popular.

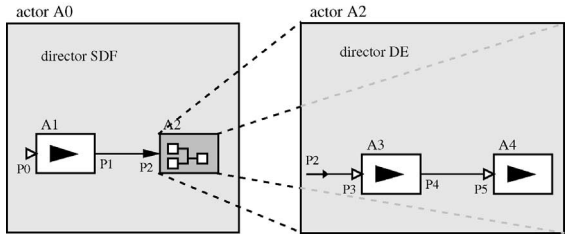

Fig. 15 *Hierarchical, heterogeneous system in Ptolemy II [15]*

Many ad-hoc solutions have been proposed, coupling two particular languages together. But there exist also a few systematic approaches to this problem.

Ptolemy II: We again take Ptolemy II as an example [15]. Consider the hierarchical system in Fig. 15. Process, or 'actor' in the Ptolemy II language, A0 consists of two other actors. A0 is an SDF model while one of the constituent actors is modelled in a different MoC, a discrete-event (DE) MoC. A DE MoC in Ptolemy II corresponds to what we have called a discrete-time MoC. In Ptolemy a process network governed by a particular MoC is called a domain. Each domain is realised by two entities: a director and a receiver. The director determines the execution order of the processes in a domain, while the receiver handles the communication between ports in the domain. When actors A1 and A2 communicate, the communication is controlled by the A0 receivers, thus it follows the SDF semantics. The SDF director also controls when A2 is invoked. However, A2 once invoked is controlled by the DE semantics. Thus, communication between P2 and P3 ports is handled by the A2 receivers and the invocation of actors A3 and A4 occurs according to the semantics of the DE MoC.

In this way, heterogeneous systems can be modelled hierarchically and any MoC can be hierarchically contained by any other MoC. The MoCs integrated in Ptolemy II include communicating sequential processes (CSP), continuous-time (CT), discrete-event (DE), distributed discrete-event (DDE), discrete-time (DT), finite-state machine (FSM), process networks (PN), synchronous data flow (SDF), synchronous/reactive (SR) and a few others [72]. Due to the separation of computation of a process from the communication with its environment, actors can be reused in different MoCs, which is called 'domain polymorphism'.

Other frameworks: Another example of a heterogeneous framework, embedded in the SystemC language, is described in [20]. The SystemC simulation semantics is extended by adding other simulation kernels that realise other MoCs. Thus, in addition to the discrete-time MoC of standard SystemC, the framework contains also an SDF, a CSP and an FSM MoC.

While the problem of simulating heterogeneous models is well addressed with Ptolemy II and Patel's SystemC libraries, cross-MoC domain analysis and verification is still an open problem. This is particularly urgent because many severe system failures occur due to a mismatch of basic assumptions between subsystems. The Ptolemy team has started to address this problem by developing a type system for communication protocols that allow for static verification of certain protocol properties [73].

A formal framework to study heterogeneous MoCs beyond simulation has been developed [16] and partially implemented in Haskell and ML [19]. As discussed above in Section 2.6, SPI [44] and FunState [45] are also aiming to

integrate heterogeneous models for cross-domain analysis and optimisation.

In summary, many theoretical and practical problems with the integration of heterogeneous MoCs are still open. If history is a guide, progress will be steady but slow, and we can expect that tools, which address cross-domain performance analysis, verification and optimisation, will only gradually be integrated into standard industrial design flows.

4.6 Why are some languages more successful than others?

Research in programming and design languages very rarely has the immediate effect, that a particular language is adopted by a large industrial community. In addition to the technical merits of a language there are so many other factors influencing its success and acceptance. The fate of a language is often tied to certain companies, applications and user communities; their success and failure is often more important than the inherent properties of a language. Having said this, we would still like to review several important technical factors that are not a necessary and direct consequence of the semantics of a language and the MoCs it supports.

4.6.1 Support for analysis and synthesis: Year after year the International Technology Roadmap for Semiconductors has listed functional validation among the main challenges in design. In its 2003 edition it reads 'Verification has become the dominant cost in the design process. On current projects, verification engineers outnumber designers, with this ratio reaching two or three to one for the most complex designs. Design conception and implementation are becoming mere preludes to the main activity of verification.' [74, page 25].

Given that validation is dominating the design, how can a language help? There are two main lines of arguments on this issue. Proponents of the first suggest that validation technology is hardly dependent on the design language. Validation methodologies, simulation engines, testbench authoring tools and even formal verification techniques can be applied to any design language to roughly the same effect. And indeed today there is a flourishing market for validation technology around and supporting traditional design languages such as VHDL and Verilog. There are a number of formal verification tools operating on VHDL and Verilog [75, 76] even though researchers have claimed for years that these languages are not suitable for formal verification. For a recent survey of the industrial usage of formal verification tools see [77, 78].

Others contend that the formalism and semantics of a language can make a huge difference in how effective analysis and verification algorithms can work. For instance when side effects are not allowed in a language, this information can be exploited by tools to establish the equivalence of two different design representations. (A function, procedure or block has a side effect when it performs an action not explicitly visible in its output. For example, when a function 'plus' not only adds two numbers and returns the result but also writes to a file and updates global variables, it is said to have side effects not apparent from the return value. Side effects are an obstacle to formal verification because the analysis has to deduce all possible side effects, which is a daunting task if variable aliasing and pointers are also present.) Otherwise the tool has to prove in every single case that there is no side effect, which may well be impossible even in the many cases where there is indeed no side effect. Also, both effective synthesis and verification techniques for VHDL and Verilog have only become possible after a proper subset has been defined and an appropriate interpretation of the syntactic structure for ease of synthesis and verification has been adopted, which is inconsistent with the simulation semantics; essentially a different MoC has been defined and embedded in VHDL and Verilog. This process has taken several years and it can be argued that the choice of VHDL and Verilog has delayed the introduction of efficient synthesis and verification tools by several years, thus widening the design productivity gap.

Even if we accept that languages are not selected only on their narrow technical merits, we can observe that exposing important properties to both designers and tools and choosing the 'right' level of detail and abstraction does have an impact on the efficiency of tools and design methodologies. The evolution of MoCs and related concepts are essentially a consequence of the search for the 'right' abstraction levels and primitive operations to be exposed to designers and tools.

Two examples may illustrate the point. Several times we have mentioned SDF as a restricted untimed MoC. However, the restrictions are well chosen because they guarantee very nice formal properties that allow us to efficiently tackle the static scheduling and buffer optimisation problem. For many applications the price of a restricted MoC is worth paying. Other examples are type systems. Type systems enforce properties on design components (variables, functions, processes, etc.) that can be statically checked. This tremendously facilitates static analysis which almost by definition is much more efficient than dynamic analysis by simulation. In fact, the SDF MoC can be viewed as a type system for processes. The type of a process is determined by the number of data items it consumes and produces during each invocation. Dynamic variations of a process type are not allowed by the SDF MoC.

Experience shows that, by defining language subsets and extensions, by enforcing modelling rules and by combining and integrating different languages, the 'right' level of detail and abstraction with the 'right' set of properties can be approximated rather well, even on the basis of established languages such as VHDL, Verilog or C. In fact, the recent popularity of the concept of computational models draws on the hope to be able to formulate essential properties independent of a language syntax and semantics and to project and embed different computational models in existing design languages. If this agenda is successful we will be able to formulate MoCs for particular purposes, such as synthesis, formal verification or system level performance analysis, and either integrate them in one language such as SystemC or in a multi-language framework, such as Ptolemy. The net result in terms of technical merits will be very significant.

4.6.2 Infrastructure: education, tools, libraries: Given the substantial investment in education and training on current languages, libraries, conventions, tools and methodologies in companies, government organisations, universities and schools, radical changes are impossible. One can argue that new languages can be learned within a few weeks and, consequently, new languages can be introduced quickly. However, even though the languages, their syntax and semantics, constitute only a tiny fraction of the overall education effort, they are, as shown in Fig. 16, at the bottom of the reversed investment pyramid, on which everything else rests.

Thus, any change and improvement can only be incremental and must not disrupt the heavy legacy of previous investments. Any more drastic change will take

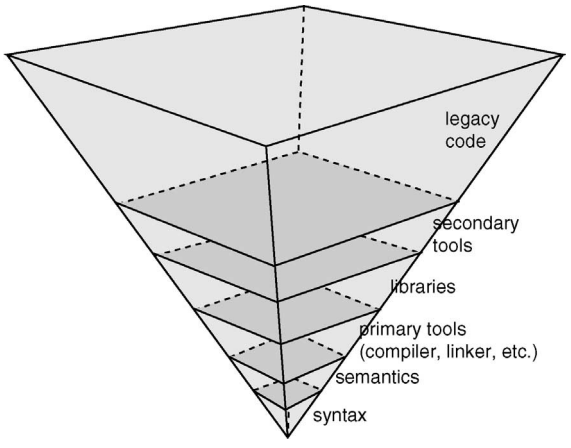

Fig. 16 *Investment pyramid*

longer and has to be prepared by establishing an education infrastructure, tools and libraries before it can compete in the market place. It has to support the upper part of the legacy pyramid at some appropriate point rather than try to replace the entire pyramid.

5 Conclusions

The importance of the semantics and properties of design languages can hardly be underestimated because they ultimately constrain what analysis, optimisation and synthesis tools can do, how well and efficient they can do it, and how productive designers are. Thus, they have a huge overall impact on the cost and performance of the designed system.

However, the demands and requirements on a design language are so versatile and contradictory that no single language or even a small set of languages, can satisfy all of them. Moreover, due to huge investment in tools, libraries and legacy code, new language technology can be introduced only gradually and slowly.

On the upside we have observed that novel language technology with their accompanying tools and methodologies can be deployed in many flexible ways. The MoC concept has shown that interesting model features and properties can be separated from the syntax and semantics of a language. This allows that a new MoC can be defined independent of a language and embedded in various existing design languages. Thus, new tools can operate on the new MoC, while existing tools operating on the design language will not see any change.

Another way to insert new language and design technology into established design flows is offered by heterogeneous frameworks that integrate different semantic domains. Heterogeneous frameworks allow integration of new languages and models easily because they only govern how the domains interact, but do not constrain the domain internal semantics and behaviour.

As a consequence, we expect that new language technology and accompanying design tools will be gradually integrated into the main stream design flows by embedding appropriate MoCs into popular languages such as SystemC, SystemVerilog, Java, etc. and by further enhancing heterogeneous frameworks with useful and specialised domains.

6 References

1 Edwards, S., Lavagno, L., Lee, E.A., and Sangiovanni-Vincentelli, A.: 'Design of embedded systems: formal models, validation, and synthesis', *Proc. IEEE*, 1997, **85**, (3), pp. 366–390
2 Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M., and Sangiovanni-Vincentelli, A.: 'System-level design: orthogonolization of concerns and platform-based design', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2000, **19**, (12), pp. 1523–1543
3 Benveniste, A., and Berry, G.: 'The synchronous approach to reactive and real-time systems', *Proc. IEEE*, 1991, **79**, (9), pp. 1270–1282
4 Grötker, T., Liao, S., Martin, G., and Swan, S.: 'System design with systemC' (Kluwer Academic Publishers, 2002)
5 Schaumont, P., Vernalde, S., Rijnders, L., Engels, M., and Bolsens, I.: 'A programming environment for the design of complex high speed ASICs'. Proc. Design Automation Conf., Anaheim, CA, USA, 1997
6 Clarke, E.M., and Wing, J.M.: 'Formal methods: state of the art and future directions', *ACM Comput. Surv.*, 1996, **28**, (4)
7 Kern, C., and Greenstreet, M.R.: 'Formal verification in hardware design: A survey', *ACM Trans. Des. Autom. Electron. Syst.*, 1999, **4**, (2)
8 McFarland, M.C.: 'Formal verification of sequential hardware: a tutorial', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 1993, **12**, (5), pp. 633–654
9 Jantsch, A., and Sander, I.: 'On the roles of functions and objects in system specification'. Proc. Int. Workshop on Hardware/Software Codesign, 2000
10 Jantsch, A.: 'Models of embedded computation', in 'Embedded systems' (CRC Press, 2004)
11 Lee, E.A., and Sangiovanni-Vincentelli, A.: 'A framework for comparing model of computation', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 1998, **17**, (12), pp. 1217–1229
12 Paul, J.M., and Thomas, D.E.: 'Models of computation for systems-on-chip', in Jerraya, A., and Wolf, W. (Eds.): 'Multiprocessor systems-on-chip' (Morgan Kaufman Publishers, 2004), Chap. 15
13 Savage, J.E.: 'Models of computation, exploring the power of computing' (Addison Wesley, 1998)
14 Taylor, R.G.: 'Models of computation and formal language' (Oxford University Press, New York, 1998)
15 Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., and Xiong, Y.: 'Taming heterogeneity—the Ptolemy approach', *Proc. IEEE*, 2003, **91**, (1), pp. 127–144
16 Jantsch, A.: 'Modeling embedded systems and SoCs - concurrency and time in models of computation: Systems on silicon' (Morgan Kaufmann Publishers, June 2003)
17 Sander, I., and Jantsch, A.: 'System modeling and transformational design refinement in ForSyDe', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 2004, **23**, (1), pp. 17–32
18 Jones, S.P.: 'Haskell 98 language and libraries' (Cambridge University Press, 2003)
19 Mathaikutty, D., Patel, H., and Shukla, S.: 'A functional programming framework of heterogeneous model of computations for system design'. Proc. Forum on Specification and Design Languages, Lille, France, September 2004
20 Patel, H.D., and Shukla, S.K.: 'SystemC kernel extensions for heterogeneous system modelling' (Kluwer Academic Publishers, Boston/Dordrecht/London, 2004)
21 Skillicorn, D.B., and Talia, D.: 'Models and languages for parallel computation', *ACM Comput. Surv.*, 1998, **30**, (2), pp. 123–169
22 Flynn, M.J.: 'Some computer organisations and their effectiveness', *IEEE Trans. Comput.*, 1972, **C-21**, (9), pp. 948–960
23 Dabney, J., and Harman, T.L.: 'Mastering SIMULINK 2' (Prentice Hall, 1998)
24 Elmqvist, H., Mattsson, S.E., and Otter, M.: 'Modelica - the new object-oriented modeling language'. Proc. 12th European Simulation Multiconference, June 1998
25 Cassandras, C.G.: 'Discrete event systems: modeling and performance analysis' (Asken Associates, 1993)
26 IEEE, IEEE Standard VHDL Language Reference Manual. IEEE, 2002
27 IEEE, IEEE Standard for Verilog Hardware Description Language. IEEE, 2001
28 Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D.: 'The synchronous data flow programming language LUSTRE', *Proc. IEEE*, 1991, **79**, (9), pp. 1305–1320
29 Winskel, G.: 'The formal semantics of programming languages' (MIT Press, 1993)
30 Berry, G.: 'The constructive semantics of pure Esterel - draft version 3'. Technical report, INRIA, 06902 Sophia-Antipolis CDX, France, 2 July 1999
31 Le Guernic, P., Gautier, T., Le Borgne, M., and Le Marie, C.: 'Programming real-time applications with SIGNAL', *Proc. IEEE*, 1991, **79**, (9), pp. 1321–1335
32 Lee, E.A., and Parks, T.M.: 'Dataflow process networks', *Proc. IEEE*, 1995, **83**, (5), pp. 773–799
33 Kahn, G.: 'The semantics of a simple language for parallel programming'. Proc. IFIP Congress 74, Stockholm, Sweden, North–Holland, 1974
34 Kahn, G., and MacQueen, D.B.: 'Coroutines and networks of parallel processes', IFIP'77 (North–Holland, 1977)
35 Lee, E.A., and Messerschmitt, D.G.: 'Static scheduling of synchronous data flow programs for digital signal processing', *IEEE Trans. Comput.*, 1987, **C-36**, (1), pp. 24–35
36 Lee, E.A., and Messerschmitt, D.G.: 'Synchronous data flow', *Proc. IEEE*, 1987, **75**, (9), pp. 1235–1245
37 Hoare, C.A.R.: 'Communicating sequential processes', *Commun. ACM*, 1978, **21**, (8), pp. 666–676
38 Milner, R.: 'A calculus of communicating systems', *Lect. Notes Comput. Sci.*, 1980, **92**
39 Milner, R.: 'Communication and concurrency' (Prentice Hall, 1989)
40 Booch, G., and Bryan, D.: 'Software engineering with Ada' (The Benjamin/Cummings Publishing Company, 1994)

41 Girault, A., Lee, B., and Lee, E.A.: 'Hierarchical finite state machines with multiple concurrency models', *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 1999, **18**, (6), pp. 742–760
42 Jantsch, A., and Bjuréus, P.: 'Composite signal flow: a computational model combining events, sampled streams, and vectors'. Proc. Design and Test Europe, Conf. (DATE), Paris, France, March 2000, pp. 154–160
43 Bjuréus, P., and Jantsch, A.: 'Modeling of mixed control and dataflow systems in MASCOT', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2001, **9**, (5), pp. 690–704
44 Ziegenbein, D., Richter, K., Ernst, R., Thiele, L., and Teich, J.: 'SPI—a system model for heterogeneously specified embedded systems', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2002, **10**, (4), pp. 379–389
45 Strehl, K., Thiele, L., Gries, M., Ziegenbein, D., Ernst, R., and Teich, J.: 'FunState - an internal design representation for codesign', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2001, **9**, (4), pp. 524–544
46 Rose, F., Leiserson, C., and Saxe, J.: 'Optimizing synthesis circuitry by retiming'. Proc. Caltech Conf. on VLSI, 1983, pp. 41–67
47 Wehn, N., Biesenack, J., Langmaier, T., Muench, M., Pilsl, M., Rumler, S., and Duzy, P.: 'Scheduling of behavioural VHDL by retiming techniques'. Proc. EuroDAC 94, 1994 pp. 546–551
48 Ashenden, P.J., Peterson, G.D., and Teegarden, D.A.: 'Designers guide to VHDL-AMS' (Morgan Kaufman, 2002)
49 Synopsys Inc.: CoCentric(R) SystemC compiler behavioral modeling guide, 2002
50 Synopsys Inc.: CoCentric(R) SystemC compiler RTL user and modeling guide, 2002
51 Vachoux, A., Grimm, C., Einwich, K.: 'SystemC-AMS requirements, design objectives and rationale'. Proc. Design Automation and Test Europe Conf., 2003
52 Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., and Simone, R.D.: 'The synchronous languages 12 years later', *Proc. IEEE*, 2003, **91**, (1), pp. 64–83
53 Halbwachs, N.: 'Synchronous programming of reactive systems' (Kluwer Academic Publishers, 1993)
54 Berry, G.: 'The foundations of Esterel', in Plotkin, G., Stirling, C., and Tofte, M. (Eds.): 'Proof, language and interaction: essays in honour of Robin Milner' (MIT Press, 1998)
55 Berry, G., and Gonthier, G.: 'The Esterel synchronous programming language: design, semantics, implementation', *Science Comput. Program.*, 1992, **19**, (2), pp. 87–152
56 Boussinot, F., and De Simone, R.: 'The ESTEREL language', *Proc. IEEE*, 1991, **79**, (9), pp. 1293–1304
57 Berry, G.: 'The Esterel v5 language primer'. Ecole des Mines and INRIA, 06565 Sophia–Antipolis, France, version v5_91 edition, July 2000
58 Harel, D.: 'Statecharts: a visual formalism for complex systems', *Sci. Comput. Program.*, 1987, **8**, (3), pp. 231–274
59 Maraninchi, F.: 'The Argos language: graphical representation of automata and description of reactive systems'. IEEE Workshop on Visual Languages, October 1991
60 Berry, G.: 'A hardware implementation of pure Esterel'. Proc. Int. Workshop on Formal Methods in VLSI Design, January 1991
61 Rocheteau, F., and Halbwachs, N.: 'Implementing reactive programs on circuits: a hardware implementation of lustre'. REX Workshop Proc., June 1992
62 Halbwachs, N., and Raymond, P.: 'Validation of synchronous reactive systems: from formal verification to automatic testing'. Proc. ASIAN'99, Asian Computing Science Conf., *Lect. Notes Comput. Sci.*, Vol. 1742, 12, Phuket, Thailand, December 1999, pp. 1–12.
63 Dennis, J.B.: 'First version of a data flow procedure language', *Lect. Notes Comput. Sci.*, 1974, **19**, pp. 362–376
64 Ashcroft, E., and Wadge, W.: 'Lucid, a nonprocedural language with iteration', *Commun. ACM*, 1977, **20**, (7), pp. 519–526
65 Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J.A.: 'Cyclo-static data flow'. Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing, 1995 pp. 3255–3258
66 Buck, J.T.: 'Scheduling dynamic dataflow graphs with bounded memory using the token flow model'. PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, USA, 1993
67 Faustini, A.A., Jagganathan, R., Ashcroft, E.A., and Wadge, W.W.: 'Multidimensional programming' (Oxford University Press, 1995)
68 Wadge, W., and Ashcroft, E.: 'Lucid, the dataflow programming language' (Academic Press, 1985)
69 Jagannathan, R.: 'Coarse-grain dataflow programming of conventional parallel computers', in Bic, L., Gaudiot, J.-L., and Gao, G. (Eds.): 'Advanced in dataflow computing and multithreading' (IEEE Computer Society Press, 1995), pp. 113–129
70 Hylands, C., Lee, E.A., Liu, J., Liu, X., Neuendorffer, S., Xiong, Y., and Zheng, H. (Eds): 'Ptolemy II - heterogeneous concurrent modeling and design in Java', volume 2, Department of Electrical Engineering and Computer Sciences University of California at Berkeley, USA, 2003. Memorandum UCB/ERL M03/29, version 3.0
71 Celoxica Limited.: Handel-C language reference manual, dk2.0 edition, 2003. RM-1003-4.0
72 Lee, E.A.: Overview of the ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley, CA, USA, July 2003
73 Lee, E.A., and Xiong, Y.: 'A behavioral type system and its application in Ptolemy II', *Form. Asp. Comput.*, 2004, **16**, pp. 210–237
74 ITRS Technology Working Group.: International Technology Roadmap for Semiconductors - design, 2003, edition, 2003
75 Habibi, A., and Tahar, S.: 'A survey: system-on-a-chip design and verification'. Technical report, Electrical and Computer Engineering Department, Concordia University, Montreal, Canada 2003
76 Roy, S.K., Ramesh, S., Chakraborty, S., Nakata, T., and Rajan, S.P.: 'Functional verification of system on chips practices, issues and challenges'. Proc. 15th Int. Conf. on VLSI Design, 2002
77 Jäger, S.: 'Technical and economical barriers and drivers for the introduction of formal methods in the verification of digital systems'. Master's thesis, Darmstadt University of Technology, Department of Electrical Engineering and Information Technology, Germany, January 2004
78 Lemire, J.-F., Regimbal, S., Bois, S., Savaria, Y., and Aboulhamid, E.M.: 'A survey on current functional verification practice'. Technical report, École Polytechniqué de Montréal, Université de Montréal, Canada, 2002