Chapter 2
Formal Modeling of Embedded Systems with Explicit Schedules and Routes

Julien Boucaron, Anthony Coadou, and Robert de Simone

A main goal of compilation is to efficiently map application programs onto execution platforms, while hiding the details of the latter to the programmer through high-level programming languages. Of course this is only feasible inside a certain range of constructs, and the judicious design of sequential programming languages and computer architectures that match one another has been a decades-long process. Now the advent of multicore processors brings radical changes to this topic, bringing forth concurrency as a key element in efficiency, both for application design and architecture computing power. The shift is mostly prompted by technological factors, namely the ability to cram several processors on a single chip, and the diminishing gains of Instruction Level Parallelism techniques used in former architectures. Still, the definition of high-level programming (and more generally, application design) formalisms matching the new era is a largely unsolved issue.

In the face of this new situation, formal models of concurrency will have to play a role. While not readily programming models, they can on the one hand provide intermediate representation formats that provide a bridge towards novel architectures. On the second hand they can also act as foundational principles to devise new programming and design formalisms, specially when matching domains have natural concurrent representation, such as dataflow signal processing algorithms. On the third hand their emphasis on communication and data transport as much as actual computations makes them fit to deal with communication latencies (to/from external memory or between computation units) at an early design stage. As a result such Models of Communication and Computation (MoCCs) may hold a privileged position, at the crossing point of executable programs and analyzable models. In the current chapter, we investigate dataflow Process Networks. They seem specially relevant as a formal framework modeling situations when execution platforms are themselves networks of processors, while applications are prominently dataflow signal processing algorithms. In addition they form a privileged setting where to express and study issues of static scheduling and routing patterns.

J. Boucaron, A. Coadou, and R. de Simone (✉)
INRIA Sophia Antipolis Méditerranée, 06902 Sophia Antipolis, France
e-mail: rs@sophia.inria.fr

S.K. Shukla and J.-P. Talpin (eds.), *Synthesis of Embedded Software: Frameworks and Methodologies for Correctness by Construction*, DOI 10.1007/978-1-4419-6400-7_2,

Outline

In the first section, we recall and place in relative positions some of the many Process Networks variants introduced in literature. We partially illustrate them on an example. Sections 2.2 and 2.3 provide more detailed formal definitions, and list a number of formal properties, some classical, some recently obtained, some as our original contribution. Section 2.2 focuses on pure dataflow models, while Section 2.3 introduces condition control switches provided it does not introduce *conflicts* between individual computations. We conclude on open perspectives.

2.1 General Presentation

2.1.1 Process Networks

As their name indicates, Process Networks (PN) consist of local processing (or computation) nodes, connected together in a network by point-to-point communication links. Generally the links are supposed to carry data flow streams. Most Process Network models are considered to be dataflow, meaning here that computations are triggered upon arrival of enough data on incoming channels, according to specific semantics. Historically the most famous PN models are Petri Nets [65], Karp-Miller's Parallel Program Schemata [50], and Kahn Process Networks [47]. The Ptolemy [39] environment is famous as a development framework based on a rich variety of PN models. More new models are emerging due to a renewed interest in such streaming models of computation, prompted by manycore concurrent architecture and dataflow concurrent algorithms in signal processing and embedded systems.

By nature PNs allow design methodologies based on assembly of components with ports into composite subsystems; hierarchical descriptions come as natural. When interconnect fabrics more complex than simple point-to-point channels are meant, they need to be realized as another component. Local components (computation nodes or composite subsystems) may contain states. Even though PNs usually accept a native self-timed semantics (which only states that computations are triggered whenever the channels data occupancy allows), a great deal of research in PN semantics consists in establishing when "optimal" schedules may be designed, and how they can be computed (and if possible statically). Such a schedule turns the dataflow semantics into a more traditional control-flow one, as in Von Neumann architectures, while it guarantees that compute operation will be performed exactly when the data arguments are located in the proper places (here channel queues instead of registers). More generally, we believe that Process Networks can be used as key formal models from Theoretical Computer Science, ready to explain concurrency phenomena in these days of embedded applications and manycore architectures. Also they may support efficient analyses and

transformations, as part of a general parallelizing compilation process, involving distributed scheduling, optimized mapping, and real-time performance optimization altogether.

2.1.2 Pure Dataflow

A good starting point for dataflow PN modeling is that of Marked Graphs [32] (also called Event Graphs in the literature). They form the simplest model, the one for which the most results have been established, and under the simplest form. For instance the questions of liveness (or absence of deadlocks and livelocks), boundedness (or executability with finite queue buffers), the question of optimal static schedules have positive answers in this model (and specially in the case of strongly connected graphs underlying the directed-communication network). Then other models of computation and communication can be considered as natural extensions of Marked Graphs with additional features. Timed Marked Graphs [68] add minimal prescribed latencies on channel transport and computation duration. Synchronous Data Flow [56] (SDF) graphs, also called Weighted Event Graphs, require that data are produced and consumed on input and output channels not individually, but along a prescribed fixed number each channel. The case of single token consumption and production is called "homogeneous" in SDF terminology, so that Marked Graphs are also called homogeneous SDF processes. Bounded Channel Marked Graphs limit the buffering capacity of channels, possibly to a value below the bound observed by safety criteria, so that further constraints of data traffic are imposed by congestion control. The recent theory of Latency-Insensitive Design [21] and Synchronous Elastic processes is attempting to introduce Bounded Channels, Timed Marked Graphs for the modeling of System-on-Chip, and the analysis of timing closure issues. It focuses on the modeling of the various elements involved by typical hardware components. In all cases the models can be expanded into Marked Graphs, but the exact relations between properties on original models and those of their reflection into Marked Graphs often need to be carefully stated. We shall consider this as we go through detailed definitions of the various such Process Networks in the next section. The syntactic inclusions are depicted in Fig. 2.1.

Any circuit in a Marked Graph supports a notion of structural throughput which is the ratio of the number of tokens involved in the circuit over its length (this is an invariant). Then, a Marked Graph contains critical circuits, and a critical structural throughput, which provides an upper bound of the allowable frequency of computa-

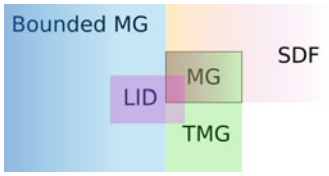


Fig. 2.1 Syntactic expressivity of Pure DataFlow

tion firings. A master result of Marked Graphs is that they can indeed be scheduled in an ultimately repetitive way which reaches the speed of this critical throughput. This result can be adapted to other extended Process Network models, and the schedules optimized against other criteria as well in addition. We shall describe our recent contributions in that direction in further sections.

2.1.3 Static Control Preserving Conflict-Freeness

The previous kinds of models could be tagged as "purely dataflow" based, as data (or rather their token abstractions) always follow the same paths. In the chapter, we shall considered extended models, where some of the nodes may select on which channels they will consume or produce respectively their input or output tokens respectively (instead of producing/consuming on each channel uniformly). Nevertheless some restrictions will remain: choices should be made according to the node current internal state, and not upon availability of tokens on various channels. Typically, the "choice between input guards" construct of CSP [44] will be forbidden, just as any synchronous preemption scheme. As a result, the extended class of Process Networks shall retain the "conflict freeness" property such as defined in Petri Nets terminology. As a result, the various self-time executions of the systems result all in the same partially ordered trace of computations. In other words, a computation once enabled must eventually be triggered or executed and cannot be otherwise disabled. Conflict freeness can also be closely associated with the notion of "confluence" (in the terms of R. Milner for process algebra), or also to the "monotonicity continuity" in Kahn Process Networks. Conflict freeness implies the important property of latency-insensitivity. The input/output semantics of the system as a whole will never depend upon the speed of travelling through channels: only the timings shall be incidentally affected.

While pure dataflow models (without conditional control) are guaranteed to be conflict-free, this property is also preserved when conditional control which bears only on internal local conditions (and not the state of connected buffer channels) are introduced. This is the case for general Kahn Process Networks, but also for so-called Boolean DataFlow [18] (BDF) graphs, which introduce two specific routing nodes for channel multiplexing and demultiplexing. We shall here name these nodes Merge and Select respectively (other authors use a variety of different names, and we only save the name *Switch* for the general feature of switching between dataflow streams according to a conditional switch pattern). This is also the case in Cyclo-Static Data Flow (CSDF) [9] graphs, where the weights allowed in SDF are now allowed to change values according to a statically pre-defined cyclic pattern; while some weights are allowed to take a null value, Merge and Select may be encoded directly in CSDF.

While the switching patterns in BDF are classically supposed to be either dynamically computed or abstracted away in some deterministic condition (as for Kahn Process Networks), one can consider the case where the syntactic simplicity of BDF

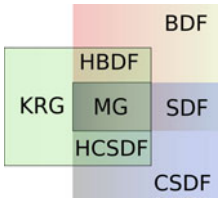


Fig. 2.2 Syntactic expressivity of MG, SDF, CSDF, BDF and KRG

(only two additional Merge/Select node types) are combined with the predictive routing patterns of CSDF. This led to our main contribution in this chapter, which we call K-periodically Routed Graphs (KRGs). Because the switching patterns are now explicit, in the form of ultimately periodic binary words (with 0 and 1 referring to the two different switching positions), the global behaviors can now be estimated, analyzed and optimized. We shall focus in some depth on the algebraic and analytic properties that may be obtained from such a combination of scheduling and routing information. The syntactic inclusions are depicted in Fig. 2.2.

To conclude, one global message we would like to pass to the reader is that Process Networks (at least in the conflict-free case) can indeed be seen as supporting two types of semantics: *self-timed* before scheduling and routing, *synchronous* (possibly multirate) after scheduling and routing have been performed. The syntactic objects representing actual schedules and routing patterns (mostly in our case infinite ultimately periodic binary words) should be seen as first-class design objects, to be used in the development process for design, analysis, and optimization altogether.

2.1.4 Example

We use the Kalman filter as supporting example through this chapter. This filter estimates the state of a *linear dynamic system* from a set of noisy measurements. Kalman filters are used in many fields, especially for guidance, positioning and radar tracking. In our case, we consider its application to a navigation system coupling Global Positioning System (GPS) and Inertial Navigation System (INS) [20].

Actually, we do not need to understand fully how the Kalman filter works, what really matters is to understand the corresponding block diagram shown in Fig. 2.3.

This block diagram shows data dependencies between operators used to build the Kalman filter. Boxes denote data flow operators: for instance, the left top box is an array of multiply-accumulates (MACs), with two input flows of 2D arrays of size $[n : n]$, and an output flow of 2D array of size $[n : n]$. The number of states (position, velocity, etc.) is denoted by n. The number of measurements (number of satellites in view) is denoted by m. Arcs denote communication channels and flow dependencies. Initial data are shown as gray filled circles.

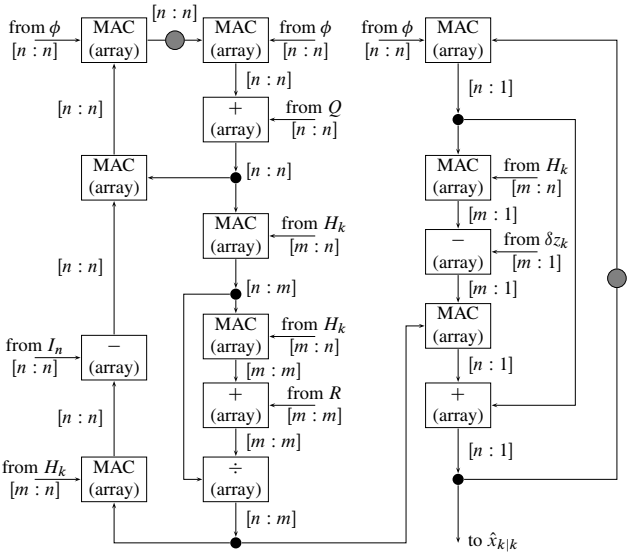


Fig. 2.3 Block diagram of the Kalman filter

We refer the interested reader to Brown et al. [16] and Campbell [20] for details on the Kalman filter. Just notice that H_k is the GPS geometry matrix, δz_k is a corrected velocity with INS data, and $\hat{x}_{k|k}$ is the estimated current state.

2.2 Pure Dataflow MoCCs

Pure dataflow MoCCs are models where the communication topology of the system is static during execution, and data follow same paths during each execution. There exists a partial order over events in such system which leads to a deterministic concurrent behavior.

First, we present the simplest pure data flow MoCC with the synchronous paradigm. After, we detail Latency-Insensitive Design. Then, we detail Marked Graphs and Synchronous Data Flow.

2.2.1 Synchrony

The synchronous paradigm is the de facto standard for digital hardware design:

- Time is driven by a clock.
- At each clock cycle, each synchronous module samples all its inputs and produces results on all its outputs.

- Computations and communications are instantaneous.
- Communications between concurrent modules are achieved through *signals* broadcast throughout the system.

Synchrony requires that designs are free of instantaneous loops (called also *causality cycles*): each loop shall contain at least one memory element (e.g., latch or flip-flop), such that when all memory elements are removed the resulting graph is acyclic. This implies a partial order of execution among signals at each instant. The behavior is deterministic with respect to concurrency.

Synchrony is control-free: control can be encoded through *if-conversions*: a control dependency is turned into a data dependency using *predicates*. For instance, "if (a) $b = c$ else $b = d$" is transformed into $b = (a \wedge c) \vee (\neg a \wedge d)$. From a digital hardware view, the previous code is equivalent to a 2-to-1 multiplexer with conditional input a, with data inputs c and d, and with data output b.

Example 1. Implementing the Kalman filter as a synchronous system is straightforward: for each box in the initial block diagram in Fig. 2.3, we assign a corresponding set of combinatorial gates; for each initial amount of data, we assign a set of memory elements (latch, flip-flop). After, we check absence of causality cycles: all circuits (in the meaning of graph theory) of the design must have at least one memory element.

In real-life, computation and communication take times. When we implement a synchronous design, performance bottlenecks are caused by highest delay combinatorial paths. Such paths are called *critical paths*, they slow down the maximum achievable clock rate.

Using multiple clocks with different rates can help implement more efficiently a design, given a set of constraints on performances and resources: such systems are sometimes called *multiclock*. A multiclock implementation introduces synchronizers for crossing multiple clock domains, synchronizers resample a single crossing clock domain.

It exists also *polychronous* implementations that introduce *logical* clock domains. Logical clock in this context represents an activation condition for a module given by a signal, such signal abstracts a *measure* of time that can be multi-form: a distance, a fixed number of elements, etc. For instance, our block diagram has different data path widths (e.g., $n : n$, $n : m$, $m : m$), we can assign to each one a logical clock corresponding to its data width. Polychronous modules are connected together using *logical* synchronizers that enable or disable the module with respect to availability of input data, output storage, etc. We refer the reader to Chaps. 1, 5 and 6 for further details.

Further Reading

Synchronous languages [7] have been built on top of synchronous and/or polychronous hypotheses. These include Esterel [66], Syncharts [3], Quartz [70],

Lustre [25], Lucid Synchrone [26, 27], and Signal [53]. They are mainly used for critical systems design (avionics, VLSI), prototyping, implementation and formal verification [10]. Tools such as SynDEx [43] have been designed to obtain efficient mapping and scheduling of synchronous applications while taking into account both computation and communication times on heterogeneous architectures.

Synchrony implicitly corresponds to the *synthesizable* subset of hardware description languages such as (System-)Verilog, VHDL or System-C. Logic synthesis tools [37] that are daily used by digital hardware designers rely on synchrony as formal MoCC. This model allows to check correctness of non-trivial optimizations [57, 72]. It enables to check design correctness using for instance formal verification tools with model-checking techniques [17, 62, 67].

Synchronous systems can contain *false* combinatorial circuits as detailed in [69]. This is a very interesting topic, since such circuits are needed to obtain digital designs with smaller delay and area. Correctness of such design is checked using tools for formal verification, based on binary decision diagrams [17, 54] or SAT solvers [38].

2.2.2 Latency-Insensitive Design

In VLSI designs, when geometry shrinks, gate delays decrease and wire delays increase due mainly to increases in resistivity as detailed in Anceau [2]. In today's synchronous chips, signals can take several clock cycles to propagate from one corner of the die to the other. Such long global wires are causing *timing closure* issues, and are not compatible with the synchronous hypotheses of null signal propagation times.

Latency-Insensitive Design (LID) [21] (also known as Synchronous Elastic in the literature) is a methodology created by Carloni to cope with such timing closure issues caused by long global wires. LID introduces a communication protocol that is able to handle any latency on communication channels. LID ensures the *same* behavior of the equivalent synchronous design, modulo timing shifts. LID enables modules to be designed and built separately, and for them to be linked together with the Latency-Insensitive protocol.

The protocol requires *patient* synchronous modules: behaviour of a patient module only depends on signal values, and not on reception times; i.e., given an input sequence, a patient module always produces the same output sequence, whatever arrival instants are. The composition of patient modules is itself a patient module as shown in Carloni et al. [22]. This requirement is a strong assumption not fulfilled by all synchronous modules that need in such case a *Shell* wrapper.

LID is built around two building blocks:

Shells. A *Shell* wraps each synchronous module (called *pearl* in LID jargon) to obtain a patient *pearl*. Shell function is two-fold: (1) it executes the pearl as soon as all input data are present and if there is enough storage to receive results in all

downward receivers (called *Relay Stations*). (2) it implements part of the latency insensitive protocol, it stores and forwards downward *backpressure* due to traffic congestion. Backpressure consists of stalling an upward emitter until a downward receiver is ready to store more data.

Relay Stations. A relay-station is a patient module: it implements the storage of results – a part of the Latency-Insensitive protocol – through a backpressure mechanism. Actually, a chain of Relay Stations implements a distributed FIFO. The minimal buffering capacity of a Relay Station is two pieces of data to avoid throughput slow-down as detailed in Carloni et al. [22]. There is at most one initial data in each Relay Station. Wires having a delay greater than one clock cycle are split in shorter ones using Relay Stations until reaching timing closure.

The core assumption in LID is that pearls can be stalled within one clock cycle. Block placements and communication latencies are not known, they have to be estimated during floor-planing, and refined further during placement and routing.

Different works [11, 13, 15, 21, 28, 33, 75, Harris, unpublished] have addressed how to implement LID Relay Stations and Shells using the *backpressure* flow-control mechanism. This solution is simple and compositional, but it suffers from a wiring overhead that may be difficult to place and route in a design.

Example 2. Implementing the Kalman filter using LID is similar to the synchronous implementation. Operators, or sets of operators – arrays of multiply-accumulates (MACs) for instance – are wrapped by Shells. Long wires are split by Relay Stations. We assume there is at least one Relay-Station in each circuit of the design for correctness purpose.

The reader should notice that LID is a simple *polychronous* system, where there is the same physical global clock and logical clocks generated by *Shells* and *Relay Stations* to execute each *Pearl*.

Further Readings

The theory is described in detail in [22]. Performance analyses are discussed in [14,15,19,23,24,29]. Formal verification of Latency Insensitive systems is described in [14,73,74]. Some optimizations are described in [14,15,19,23,24,29]. LID is also used in network fabrics (Networks on Chips), as described in [34, 41, 45]. Handling of variable latencies is described in [5, 6, 30].

2.2.3 Marked Graphs

Marked Graphs (MG) [32] are a *conflict-free* subset of Petri nets [65], also known as *Event Graphs* in the literature. MGs are widely used, for instance in discrete events simulators, modeling of asynchronous designs or job shop scheduling.

We briefly introduce its definition and operational behavior. We introduce results on deadlock freeness, and results on static schedulability and maximum achievable throughput. Then, we briefly recall Timed Marked Graphs, and we recall the case of Marked Graphs with bounded capacities on places. Then, we show the impact of such capacities on the throughput of a MG with capacities. After, we show two optimizations on MG. First, an algorithm on MGs with capacities that computes the capacity of each place such that the graph reaches the maximum achievable throughput of its equivalent MG without capacities. Next, we provide a throughput-aware algorithm called *equalization* that slows down the fastest parts of the graph while ensuring the same global throughput. This algorithm provides *best return on investment* locations to alter the schedule of the system. This can allow for instance to postpone an execution of a task to minimize power hotspots.

Definition 1 (Marked Graph). A Marked Graph is a quadruplet $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0 \rangle$, such that:

- $\mathcal{N}$ is a finite set of *nodes*.
- $\mathcal{P}$ is a finite set of *places*.
- $\mathcal{T} \subseteq (\mathcal{N} \times \mathcal{P}) \cup (\mathcal{P} \times \mathcal{N})$ is a finite set of edges between nodes and places.
- $M_0 : \mathcal{P} \to \mathbb{N}$ is the function that assigns an initial marking (quantity of data abstracted as *tokens*) to each place.[1]

In a MG, nodes model processing components: they can represent simple gates or complex systems such as processors. Each *place* has exactly one input and one output and acts as a point-to-point communication channel such as a FIFO.

The behavior of a MG is as follows:

- When a computation node has at least one token in each input place, then it *can* be executed. Such node is said *enabled* or *activated*.
- If we execute the node, then it consumes one token in each input place and produces one token in each output place.

A MG is deterministic in case of concurrent behaviour.
A MG is *confluent*: for all sets of activated nodes, the firing of any node does not remove another node from this subset than itself. This leads to a partial order of events.

We recall some key results on Marked Graphs from the seminal paper of Commoner et al. [32], where associated proofs can be found.

Lemma 1 (Token count). *The token count of a circuit does not change by node execution.*

Theorem 1 (Deadlock-freeness). *A marking is deadlock-free if and only if the token count of every circuit is positive.*

[1] We recall that $\mathbb{N} = \{0, 1, 2, \dots\}$ and $\mathbb{N}^* = \mathbb{N} \setminus \{0\}$.

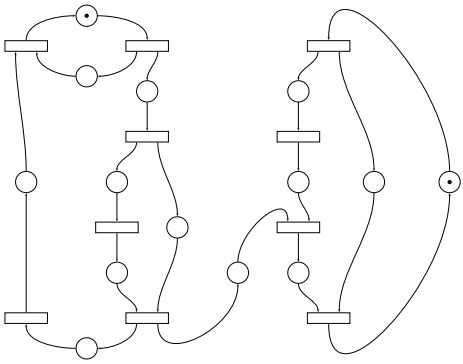


Fig. 2.4 Coarse-grain abstraction of the Kalman filter as a MG

Example 3. Figure 2.4 shows a basic model of the Kalman filter as a MG. The graph has five circuits: three in the left strongly connected component, and two in the right one. They all contain one token: the system is deadlock free.

2.2.3.1 Static Schedulability and Throughput

Now, we recall useful results on static schedulability of MG and how to compute its throughput. Static schedulability is an interesting property that enables to compute a performance metric at compilation time.

The *As Soon As Possible* (ASAP) firing rule is such that when a node is enabled, it is executed immediately. The ASAP firing rule is also known as *earliest* firing rule in literature. Since a MG is confluent, any firing rule will generate a partial order of events compatible with the ASAP firing rule. The ASAP firing rule generates the fastest achievable throughput for a system.

Definition 2 (Rate). Given a circuit $\mathcal{C}$ in a MG. We denote the node count of the circuit $\mathcal{C}$ as $L(\mathcal{C})$, and the token count of the circuit $\mathcal{C}$ as $M_0(\mathcal{C})$. We denote the *rate* of the circuit $\mathcal{C}$ as the ratio $\frac{M_0(\mathcal{C})}{L(\mathcal{C})}$.

Notice that circuits of a strongly connected component have *side effects* on each others: circuits with low rates slow down those with higher rates.

Theorem 2 (Throughput). *The throughput of a strongly connected graph equals the minimum rate among its circuits. The throughput of an acyclic graph is* 1.

Proof. Given in [4].

Strongly connected graphs reach a periodic *steady regime*, after a more-or-less chaotic initialization phase. These MGs are said k-periodic and are statically schedulable as shown in [4].

Definition 3 (Critical circuit). A circuit is said critical if its rate equals the graph throughput.

2.2.3.2 Timed Marked Graphs

Timed Marked Graphs (TMG) are an extension of MG introduced by Ramchandani [68] where *time* is introduced through weights on places and nodes. They represent abstract amounts of time needed to move the set of tokens from inputs to outputs.

Definition 4 (Timed Marked Graph). A Timed Marked Graph is a quintuplet $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0, L \rangle$, such that:

- $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0 \rangle$ is a Marked Graph.
- $L : \mathcal{N} \cup \mathcal{P} \rightarrow \mathbb{N}$ is a function that assigns a latency to each place and computation node.

Ramchandani showed that TMGs have the same expressivity as MGs [68]. He provides a simple transformation from a TMG to an equivalent MG.

The transformation is shown in Fig. 2.5 for both a place with a latency b, and a node with a latency c. The place of latency b is expanded as a succession of the pattern of a node followed by a place with a unitary latency. The succession of this pattern has the same latency b. The transformation for the node of latency c is similar, with a succession of the pattern of a unitary latency place followed by a node.

All theoretical results on MG hold for TMGs.

2.2.3.3 Bounded Capacities on Places

Real-life systems have finite memory resources. We introduce place *capacities* in MG to represent the maximal number of data that can be stored in a place: we add to Definition 4 the function $K : \mathcal{P} \rightarrow \mathbb{N}$ that assigns to each place the maximum number of tokens it can hold.

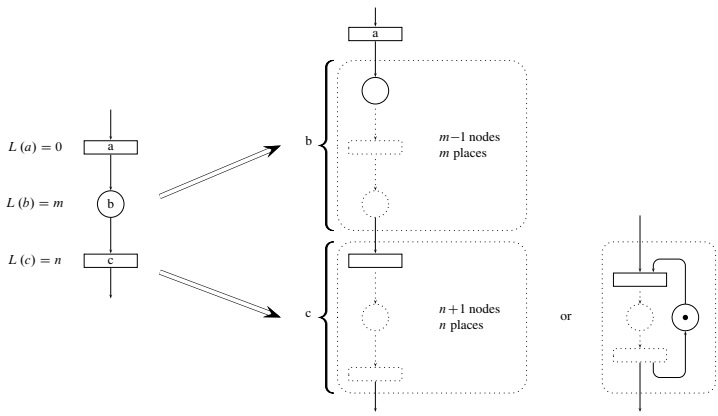


Fig. 2.5 Example of latency expansion

A MG with capacities can be transformed in an equivalent one without capacities through introduction of *complementary* places [4]: a place p_1 with a capacity $K(p_1)$ from node n_1 to node n_2 is equivalent to a pair of places p_2 and $\overline{p_2}$, where p_2 is from n_1 to n_2, and $\overline{p_2}$ is from n_2 to n_1. We write $\overline{\mathcal{P}}$ for the set of complementary places of $\mathcal{P}$ and for $\overline{\mathcal{T}}$ their corresponding arcs.

Definition 5 (Complemented graph). Given $\mathcal{G} = \langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0, K \rangle$ a connected MG with finite capacities, we build the complemented graph $\mathcal{G}' = \langle \mathcal{N}', \mathcal{P}', \mathcal{T}', M_0' \rangle$ as follows:

$$\begin{aligned} \mathcal{N}' &= \mathcal{N}, \\ \mathcal{P}' &= \mathcal{P} \cup \overline{\mathcal{P}}, \\ \mathcal{T}' &= \mathcal{T} \cup \overline{\mathcal{T}}, \\ \forall\, p \in \mathcal{P},\ M_0'(p) &= M_0(p), \text{ and} \\ M_0'(\overline{p}) &= K(p) - M_0(p). \end{aligned}$$

The hint for the proof of correctness of this transformation is as follows: when we introduce a complementary place, we introduce a new circuit holding the place and the complementary one. This new circuit has a token count equal to the capacity of the original place. Since token count is invariant by node execution, both places cannot have more tokens than the capacity of the original place.

In a regular MG, a node can produce data when it has one token on all inputs. It stops only when an input is missing. In MG with finite capacities, nodes are awaiting on inputs and also on outputs. This means that a finite capacity graph can slow down the maximum achievable throughput of the topological equivalent regular graph.

Example 4. Initially in Fig. 2.6, the graph has two circuits: the left circuit has a rate of $5/5 = 1$, and the right one has a rate of $4/5$. The capacity of each place equals two. After application of the transformation, the transformed graph without capacities has more circuits due to complementary places. In particular, there is

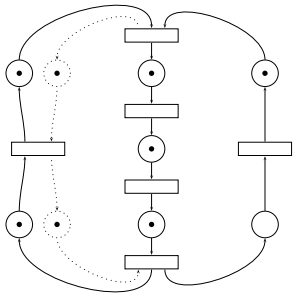


Fig. 2.6 A strongly connected TMG with unitary latencies whose throughput is limited to $3/4$ due to 2-bounded places. Plain places belong to the graph with finite capacities; we only show the most significant complementary places with *dotted lines*

a new circuit whose throughput equals 3/4. This circuit slows down the system throughput, due to lack of capacity on places.

Notice that the property of acyclicity loses its soundness when considering capacity-bounded graphs. Actually, introducing complementary places creates circuits. An acyclic graph can have a throughput lower than 1. The throughput of a connected graph with finite buffering resources equals the minimum rate among the circuits of its complemented equivalent.

Corollary 1 (Throughput w.r.t. capacities). *Let $\mathcal{G} = \langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0, K \rangle$ be a connected MG with finite capacities, and $\mathcal{G}' = \langle \mathcal{N}', \mathcal{P}', \mathcal{T}', M_0' \rangle$ its complemented graph. The maximum reachable throughput $\theta(\mathcal{G})$ of $\mathcal{G}$ is*

$$\theta(\mathcal{G}) = \min_{\mathcal{C} \in \mathcal{G}'} \left(\frac{M_0'(\mathcal{C})}{N'(\mathcal{C})}, 1 \right). \tag{2.1}$$

Proof. Given in [4].

2.2.3.4 Place Sizing

Previously, we have seen how to compute the throughput of a MG and the throughput of a MG with bounded capacities. The first one only depends on communication and computation latencies, while the second one can be lower due to capacity bounds. Now, we address the issue of place sizing to reach the maximal throughput of the graph. Our goal is to minimize the overall sum of place capacities in the MG. We state it as the following Integer Linear Programming (ILP) problem (a similar ILP formulation for the same problem is given in Bufistov et al. [19]):

$$\text{minimize} \sum_{p \in \mathcal{P}} k(p), \tag{2.2}$$

where $\mathcal{P}$ is the set of places.

It is subject to the following set of constraints in the complementary graph: for each circuit $\mathcal{C}$ that contains at least a complementary place,

$$\frac{\sum \text{tokens} + \sum k}{\sum \text{latencies}} - \frac{\sum_{\text{critic}} \text{tokens}}{\sum_{\text{critic}} \text{latencies}} \geq 0, \tag{2.3}$$

where the left part of the constraint is the throughput of the circuit $\mathcal{C}$ we consider, and the right part is the maximum achievable throughput of the graph without capacities. This program states that we want to minimize global capacity count: we add additional capacities to places through $\sum k$ in the marking of complementary places found in each circuit $\mathcal{C}$, until we reach the maximal throughput of the graph. We assume $k \geq 0$.

Now, we provide the algorithm to compute minimum capacities for each place in order to reach the maximum throughput of the system:

1. Compute the maximum throughput of the *not-complemented* graph using, for instance, the Bellman–Ford algorithm, or any minimum cycle mean algorithm [36, 49].
2. Build the *complemented* graph using previous transformation in Definition 5.
3. Enumerate all directed circuits having at least one complementary place in the *complemented* graph. We can use Johnson's algorithm [46] with a modification: when a circuit is found, we check that the circuit contains at least one *complementary* place.
4. Build and solve the previous formulation of the ILP.

As the reader understands this optimization does not come for free. Additional resources are needed to reach the maximum throughput of the system.

2.2.3.5 Equalization

Now, we recall an algorithm called *equalization* [14]. Equalization slows down as much as possible the fastest circuits in the graph, while maintaining the same global system throughput. This transformation gives hints on potential *slack*. Such slack can be used to add more pipeline stages while ensuring the same performance of the whole system. It can be used to postpone an execution, for instance to smooth dynamic power and flatten temperature hot-spots.

We provide a *revised* equalization that minimizes the amount of additional latency to minimize resource overhead. *Additional latency* here is a dummy node followed by a single place attached to an existing place. The problem is stated as the following Integer Linear Program:

$$\text{maximize} \sum_{p \in \mathcal{P}} \text{weight}\,(p) \times a\,(p)\,, \tag{2.4}$$

where $a\,(p)$ are additional latencies assigned to each place p, and $weight(p)$ is the sum of occurrences of place p in all circuits of the graph. We have the following set of constraints assigned to each non-critical circuit $\mathcal{C}$:

$$\frac{\sum \text{tokens}}{\sum \text{latencies} + \sum a} - \frac{\sum_{\text{critic}} \text{tokens}}{\sum_{\text{critic}} \text{latencies}} \geq 0, \tag{2.5}$$

where the left part of the constraint is the fast circuit that we slow down with an amount of additional latencies a, and the right part is the critical throughput of the MG. We also ensure that we have $a(p) \geq 0$. The weight found in the objective is used to force to choose places shared by different circuits to minimize additional latencies.

The algorithm is as follows:

- Compute the maximum throughput of the MG using the Bellman–Ford algorithm, or any minimum cycle mean algorithm
- Enumerate all directed circuits using for instance Johnson's algorithm
- Build and solve previous formulation of the ILP

Links with Other MoCCs

MG is a *self-timed* system; there is no global clock as in Synchrony, each component has its own clock. MG is a polychronous system where each component clock is driven by presence of tokens on inputs and its firing rule.

Synchrony is a special case of MG running under an ASAP firing rule. The transformation from Synchrony to MG is as follows: we assign to each node in the MG a directed acyclic graph of combinatorial gates; we assign to each place a corresponding memory element (flip-flop, latch) and we put an initial token. The obtained MG behaves as a synchronous module, at each instant all nodes are executed as in the synchronous model.

LID is somehow a special case of MG with bounded capacities. A Shell behaves as a node does: both require data on all their inputs and enough storage on their outputs to be executed. Relay-stations carry and store data in-order as places. But not all implementations of LID behave like a MG, especially when there are some optimizations done on the *Shell* where we can use its buffering capacity to extend further the capacity of input Relay Stations: this can avoid further slow-down of the system.

2.2.4 Synchronous Data Flow

Synchronous Data Flow (SDF) also known as *Weighted Event Graphs* in the literature, has been introduced by Lee and Messerschmitt [55,56]. SDF is a generalization of previous Marked Graphs, where a strictly positive *weight* is attached to each input and output of a node respectively. Such weight represents how many tokens are consumed and produced respectively, when the node is executed.

SDF is a widely used MoCC, especially in signal processing. It describes performance critical parts of a system using processes (actors) with different constant data rates. SDF is confluent and deterministic with respect to concurrency.

The *weight* in SDF raises the problem of deciding whether places are bounded or not during execution: if not, there is an infinite accumulation of tokens. Lee and Messerschmitt [56] provides an algorithm to decide if a SDF graph is bounded, through *balance equations*. The idea is to check if there exists a static schedule where token production equals token consumption.

Balance equations are of the form:

$$\text{weight}(n) \times \text{executions}(n) - \text{weight}(n') \times \text{executions}(n') = 0, \tag{2.6}$$

where n and n' are respectively the producer and the consumer of the considered place, and executions are unknown variables. To solve this equation, we can use a Integer Linear Programming solver, or better yet, a Diophantine equation solver [64].

If there is no solution, then there exists an infinite accumulation of tokens in the SDF graph. Otherwise, there exists an infinite number of solutions and the solver gives us one solution. This solution holds the number of occurrences of execution for each node during the period of the schedule.

After boundedness check, we can perform a symbolic simulation to compute a static schedule using the previous solution. We can check for deadlock-freedom and compute place capacities.

Example 5. Figure 2.7 represents a SDF graph corresponding to the Kalman filter (Fig. 2.3), where we consider $n = 8$ states and $m = 5$ measurements. Balance equations solving gives the trivial solution of one execution for each node during the period.

Further Readings

Several scheduling techniques have been developed to optimize different metrics such as throughput, schedule size, or buffer requirement [8, 42]. Some results on deadlock-freeness are found in Marchetti et al. [60, 61].

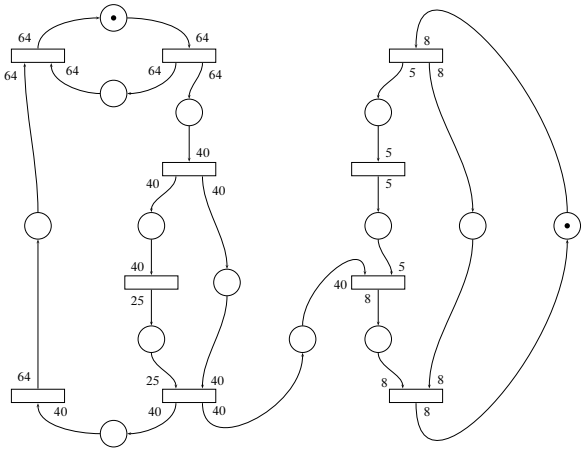


Fig. 2.7 SDF graph of the Kalman filter ($m = 5, n = 8$)

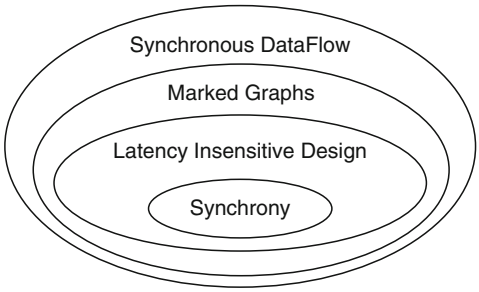


Fig. 2.8 Expressiveness of Pure DataFlow MoCCs

Retiming is a commonly used optimization in logic synthesis [57]. In case of SDF, retiming is used to minimize the cycle length or maximize the throughput of the system [59,63]. The correctness of the retiming algorithm uses a useful transformation from a bounded SDF graph to an equivalent MG described in Bhattacharyya et al. [8].

StreamIt [1, 48, 52, 71, 76] is a language and compiler for stream computing where SDF is the underlying MoCC. StreamIt allows exposing concurrency found in stream programs and introducing some hierarchy to improve programmability. The StreamIt compiler can exploit this concurrency for efficient mapping on multicore, tiled and heterogeneous architectures.

Summary

This section provides an overview of pure dataflow MoCCs. Expressiveness of pure dataflow MoCCs is described in Fig. 2.8. Such MoCCs have the following properties:

- Deterministic behavior with respect to concurrency.
- Decidability on bounded buffers during execution: structural in case of Latency-Insensitive Design and Marked Graphs; using balance equations algorithm in case of Synchronous Data Flow.
- Static communication topology, that leads to a partial order of events.
- Static schedulability: the schedule of a bounded system can be computed at compilation time with needed buffer sizes.
- Deadlock-freedom checks using a structural criteria in case of Latency-Insensitive Design and Marked Graph; or using bounded length symbolic simulation in case of Synchronous Data Flow.

2.3 Statically Controlled MoCCs

Previously, we introduced pure dataflow MoCCs where the communication topology is *static* during execution. Such topology does not allow the reuse of resources to implement different functionality.

Now, we introduce *statically controlled* MoCCs where the communication topology is dynamic and allows resources to be reused. *Static* in this context means that the control (routing of data in our case) is known at compilation time.

The main interest in such statically controlled MoCCs is that it allows greater expressivity while conserving all key properties of pure dataflow MoCCs. Such MoCCs are deterministic and confluent. We can decide at compilation time if buffers are bounded during execution. We can also check if the system is deadlock free. We can statically schedule a bounded system to know its throughput and the size of its buffers. Such MoCCs also generate a partial order on events: for instance, this allows us to check formally if a transformation applied on an instance of such MoCC is correct.

We briefly recall the MoCC Cyclo-Static Dataflow (CSDF). Then, we detail K-periodically Routed Graphs (KRG).

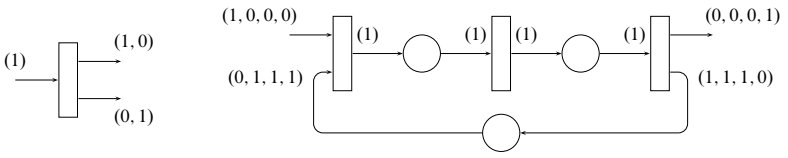
2.3.1 Cyclo-Static Dataflow Graphs

Cyclo-Static DataFlow (CSDF) [9, 40] extends SDF such that weights associated to each node can change during its execution according to a repetitive cyclic pattern. Actually, CSDF introduces *control modes* in SDF, also called *phases* in the literature.

The behavior of a CSDF graph is as follows:

- A node is *enabled* when there is at least the amount of tokens given by the *index* on the repetitive cyclic pattern associated on each input.
- When a node is *executed* then it consumes and produces respectively the amount of tokens given by the *index* on each cyclic pattern according to the associated input and output respectively. Finally, we increment the *index* of each pattern modulo its length to prepare the next execution.

Example 6. Figure 2.9a shows how we can model a branch node in CSDF. When we start the system, every *index* is 0. When this node is executed, it always consumes one input token since the pattern is (1). But this is different for outputs: at first execution, it produces only one token on the upper output since the index of the



(a) Branching **(b)** Loop

Fig. 2.9 Routing examples in CSDF

pattern is 1, and the lower output index is 0. At the end of this first execution, we increment all indices of the node.
Now, we execute this node a second time. It does not change for the input still having the index on 1, but it changes for outputs: now, the upper output produces 0 tokens, and the lower one produces 1 token. We increment all index of the node, and we get back to the same behaviour as in the first execution.

Figure 2.9b shows how to model a simple *for* loop: when an input token arrives on the upper left input, then we start the loop where the token loops three times in the circuit before exiting.

CSDF is confluent and deterministic with respect to concurrency. It can be decided if memories are bounded, and if so the CSDF graph is statically schedulable. Static routing can also be modeled using CSDF, although it is implicit: routing components are denoted as computation nodes.

2.3.2 *K*-Periodically Routed Graphs

In CSDF, it is difficult to check correctness of a transformation applied on the topology of a graph. The issue is that we do not know clearly whether a node performs a computation, whether it routes data, or a combination of both.

KRG has been introduced to tackle this issue. KRG is inspired by CSDF; the key difference is that nodes are split in two classes as done in BDF: one for stateless routing nodes, and another one for computation nodes. In KRG, only routing nodes have cyclic repetitive patterns as in CSDF, while computation nodes behave as in MG. In KRG, there are two kinds of routing nodes called *Select* and *Merge*, akin to demux and mux. Such routing nodes have at most two inputs or two outputs respectively, this allows the use of binary words to describe the cyclic repetitive pattern instead of positive integers.

This section will be longer and more technical than previous ones. After the definition of KRG, we show decidability of buffer boundedness through an abstraction to SDF. Then, we show how we can check deadlock-freedom through symbolic simulation, and also through the use of a *dependency graph*. Such dependency graph leads to a similar transformation from a SDF graph to a MG: such transformation is needed for instance to apply a legal *retiming* in case of SDF. Finally, we present KRG routing transformations on routing nodes, and show they are legal through the use of *on* and *when* operators applied on routing patterns.

Before providing the definition of a KRG, we recall some definitions borrowed from the n-synchrony theory [31]:

- $\mathbb{B} = \{0, 1\}$ is the set of binary values.
- $\mathbb{B}^*$ is the set of finite binary words.
- $\mathbb{B}^+$ is the set of such words except the empty word ε.
- $\mathbb{B}^\omega$ is the set of infinite binary sequences.
- We write $|w|$ for the length of the binary word w.

- We write $|w|_0$ and $|w|_1$ respectively for the number of occurrences of 0 and 1 in w.
- We write w_i for the ith letter of w; we note $w_{\text{head}} = w_1$ and w_{tail} such that $w = w_{\text{head}}.w_{\text{tail}}$.
- We write $[w]_i$ for the position of the ith "1" in w.
  For instance, $[0101101110]_4 = 7$. As a rule, $[0]_1 = +\infty$.
- A sequence s in $\mathbb{B}^\omega$ is said *ultimately periodic* if and only if it is of the form $u.v^\omega$, where $u \in \mathbb{B}^*$ and $v \in \mathbb{B}^+$. We call u the *initial* part and v the *steady* (or periodic) part.
- We write $\mathbb{P}_p^k$ for the set of ultimately periodic binary sequences (or *k-periodic sequences*), with a steady part of p letters, including k occurrences of 1. We call k the *periodicity* and p the *period*.
- We write $\mathbb{P}$ for the set of all ultimately periodic sequences.

Definition 6 (*K*-periodically Routed Graph). A *K-periodically Routed Graph* (KRG) is a quintuple $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0, R \rangle$, where:

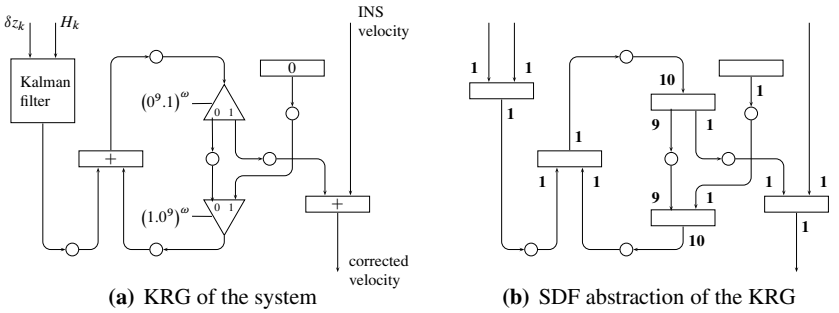
- $\mathcal{N}$ is a finite set of nodes, divided in four distinct subsets:
  - $\mathcal{N}_x$ is the set of *computation* nodes.
  - $\mathcal{N}_c$ is the set of stateless *copy* nodes. Each copy node has exactly one input and at least two outputs.
  - $\mathcal{N}_s$ is the set of *select* nodes. Each select node has exactly one input and two outputs.
  - $\mathcal{N}_m$ is the set of *merge* nodes. Each merge node has exactly two inputs and one output.
- $\mathcal{P}$ is a finite set of *places*. Each place has exactly one producer and one consumer.
- $\mathcal{T} \subseteq (\mathcal{N} \times \mathcal{P}) \cup (\mathcal{P} \times \mathcal{N})$ is a finite set of edges between nodes and places.
- $M_0 : \mathcal{P} \to \mathbb{N}$ is a function assigning an initial marking to each place.
- $R : \mathcal{N}_s \cup \mathcal{N}_m \to \mathbb{P}$ is a function assigning a routing sequence to each select and merge node.

Places and edges model point-to-point links between nodes. A KRG is *conflict free*.

Given a place p, we denote $\bullet p$ and $p\bullet$ its input and output node respectively. Similarly for a node n, we denote $\bullet n$ and $n\bullet$ its set of input places and output places.

The behavior of a KRG is as follows:

- A computation node has the same behavior as in MG: when a token is present on each input place, the node is *enabled*. An enabled node can be *executed*. When executed it consumes and produces one token on each input place and output place respectively.
- A copy node is a special case of stateless computation node where tokens on the input place are duplicated on each output place.



(a) KRG of the system **(b)** SDF abstraction of the KRG

Fig. 2.10 Correcting INS velocity using GPS position

- A select node splits a token flow in two parts. It is enabled when there is a token on its unique input place; when executed it consumes the input token and it produces a token on one of its output places. The output of production is given by the *index* of the routing pattern: if the index points to a 1 or 0 respectively, then it produces on the output labeled 1 or 0 respectively. At the end of the execution, the index of the routing pattern is incremented modulo the length of the steady part of the routing pattern. See Fig. 2.9 and Example 6 for a similar select node in CSDF.
- A merge node interleaves two token flows into one. It is enabled when a token is available on the input place pointed to by the index of the routing pattern. When executed, the input token is consumed and routed to the output place. At the end of the execution, the index is incremented modulo the length of the steady part of the routing pattern.

Example 7. We come back to our Kalman filter example. Now we integrate it into a navigation system that computes the velocity of a vehicle, as shown in Fig. 2.10a. Inertial Navigation Systems suffer from integration drift: accumulating small errors in the measurement of accelerations induces large errors over time. INS velocity is corrected at a low rate (1 Hz) with integrated error states based on GPS data at higher rate (10 Hz).

Integrating ten error states is performed with a for loop: at the beginning of each period, the merge node consumes a token on its right input, when result is reset to 0. Then, data loop through the left input of the merge node for the next nine iterations. Conversely, the select node is the break condition: the nine first tokens are routed to its left output, while the tenth exits the loop. Its SDF abstraction is given in Fig. 2.10b.

2.3.2.1 Buffer Boundedness Check

As in CSDF, buffer boundedness is decided through an SDF abstraction. We solve balance equations of the SDF graph in order to compute node firing rates and buffer bounds.

Given a KRG, the corresponding SDF graph is constructed as follows:

- The communication topology of the graph is identical for places and edges.
- Computation and copy nodes are abstracted as SDF nodes with a weight of 1 on all inputs and all outputs.
- Select and merge nodes are also abstracted as SDF nodes where the weight of each labeled input or output corresponds to the number of occurrences of the label found in the steady part of the routing pattern. In case of unlabeled input or output, the associated weight is the length of the steady part of the routing pattern.

In Fig. 2.10, we show the result of the application of this abstraction reduction. Solving balance equations to show that the KRG is bounded is left as an exercise.

2.3.2.2 Deadlock-Freedom Check

Given a bounded KRG deadlock-freedom only depends on its initial markings. We can use a bounded length symbolic simulation with the following halting conditions:

- *Deadlock*: Simulation ends when no node is enabled.
- *Periodic and live behavior*: Simulation ends when we found a periodic behavior: we get back to an already reached marking at index x in the list of reached markings, then we check for equal markings:
  $[x : x + j - 1] \equiv [x + j : x + j + j - 1]$, where j is the distance to the new marking in the list of reached markings. If we have equal markings, then we have found a periodic behavior $[x : x + j - 1]$ with an initialization from $[0 : x - 1]$.

2.3.2.3 Static Schedulability and Throughput

KRG is *confluent* due to the conflict freedom property on places: when a node is *enabled*, it remains enabled until executed.

We can introduce latency constraints in our model; we call the result Timed KRG. We define a new function associated to a KRG:

$L : \mathcal{N} \cup \mathcal{P} \rightarrow \mathbb{N}$ is a latency function, assigning a positive integer to places and nodes. Latencies of copy, merge or select nodes are supposed to be null. N.B.: We assume that there is no circuit where the sum of latencies is null, otherwise we have a combinational cycle as in synchrony.

Computation and communication latencies are integer and possibly non-unit. However, the synchronous paradigm assumes evaluations at each instant. This problem is solved by graph expansion to KRG, in a similar way as in TMG [68]: for each node with non-null latency (resp. place with latency higher than 1), it is split using intermediate nodes with null latency and places with unit latency. A non-reentrant node can be modeled with a data dependency (or loop) [4], as shown in Fig. 2.5.

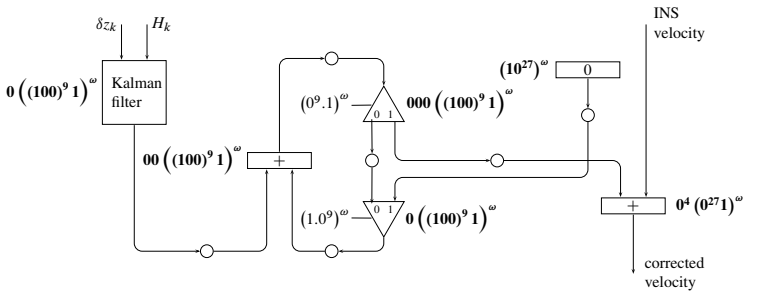


Fig. 2.11 Example of KRG scheduling, in *bold font*. We have arbitrarily set place latencies to 1 and node latencies to 0

Timed KRG have the same expressivity as KRG.

Similarly as found in MG, we introduce an ASAP firing rule such that when any node is enabled, it is executed immediately. Since a KRG is confluent any firing rule will generate a partial order of event compatible with the ASAP firing rule. The ASAP firing rule generates the fastest achievable throughput for a given KRG.

Definition 7. The *rate* of a binary word w is defined as rate $(w) = 1$ if $w = \varepsilon$, and rate $(w) = |w|_1/|w|$ otherwise. The rate of a k-periodic sequence $s = u.v^{\omega}$ equals the rate of its steady part: rate $(s) =$ rate (v).

Definition 8. The *throughput* of a node is the rate of the steady part of its schedule. The throughput of a KRG is the list of all input and output node throughputs.

Unlike Marked Graphs, the throughput of a KRG may differ from the throughput of its slowest circuit (or path).

Example 8. We introduce latencies on the KRG in Fig. 2.11. We assume here that place latency equals 1, and node latency equals 0. Bold sequences next to each node are their respective schedules, with an ASAP firing rule and with respect to boundedness constraints.

2.3.2.4 Dependency Analysis

Dependency analysis [51] is one of the most useful tool for compilation and automatic parallelization [35]. Some techniques can be transposed to KRG for the problematic of token flow dependencies.

Dependency analysis in our case can provide an *expansion* from a bounded KRG to a MG with the same behavior. Such transformation is useful to check if a transformation applied on a KRG is legal or not. This transformation can also be used to show if two instances of a KRG have the same behavior modulo timing shifts.

First, we introduce some definitions on relations and ordering properties on token flows. Next, we explain data dependencies in token flows and their representations as dependency graphs. Then, we show how to build a behavior-equivalent MG.

Definition 9 (Token flow). Let $\mathcal{E}$ be the set of tokens passing through a node n or a place p of a KRG within a period. Sequential productions then consumptions define a total order $<_{\text{seq}}$ on $\mathcal{E}$. Token flow a, passing through n or p, is a sequence $a_1.a_2...$ such that for all $a_i, a_j \in \mathcal{E}, i < j$ if and only if $a_i <_{\text{seq}} a_j$.

Definition 10 (Prefix). The prefix of length n of a binary word or sequence w is defined such that $\text{pre}(w, n) = w_1 \dots w_n$.

Definition 11 (Relations between flows). Nodes and places of a KRG apply transformations on their input flows to produce output flows. Relations between such flows are as follows:

- Let $p \in \mathcal{P}$. a is the input flow and b is the output flow such that: $b = c.a$, where c is the prefix caused by initial tokens in p, and $|c| = M_0(p)$.
- Let $n \in \mathcal{N}_x$. a is the input flow and b is the output flow such that: $b = g(a_1).f(a_2)...$, where g is the function applied by n to token data.
- Let $n \in \mathcal{N}_c$. a is the input flow and b is the output flow such that: $b = a$.
- Let $n \in \mathcal{N}_s$. a is the input flow, and b and c are the zeroth and first output respectively. There is a sequence of sub words of a, such that $a = a_1.a_2.a_3.a_4...$ and $b = a_1.a_3...$ and $c = a_2.a_4...$, chosen in a such way that the $|a_1|$th first letters of $R(n)$ are only 0s, the $|a_2|$th next ones are only 1, etc.
- Let $n \in \mathcal{N}_f$. a and b are the zeroth and first input respectively, and c is the output flow such that: $c = a \sqcup\!\sqcup_{R(f)} b$.

We recall that $\sqcup\!\sqcup$ is the shuffle product of two words or sequences, recursively defined by the equations:

$$w \sqcup\!\sqcup \varepsilon = \varepsilon \sqcup\!\sqcup w = \{w\}, \tag{2.7}$$

$$a.v \sqcup\!\sqcup b.w = \{a.(v \sqcup\!\sqcup b.w)\} \cup \{b.(a.v \sqcup\!\sqcup w)\}. \tag{2.8}$$

We write $\sqcup\!\sqcup_u$ for the restriction of the shuffle product according to sequence u; if $c = a \sqcup\!\sqcup_u b$, then:

$$\forall i \in \mathbb{N}^*, \quad c_i = \begin{cases} a_{|\text{pre}(\overline{R(f)},i)|_1} & \text{if } u_i = 0, \\ b_{|\text{pre}(R(f),i)|_1} & \text{if } u_i = 1. \end{cases} \tag{2.9}$$

Definition 12 (Elementary order relations). We associate with each token going through a node (place resp.) a pair $(i, j) \in \mathbb{N}^* \times \mathbb{N}^*$, where i is the token position in the node input flow (place input flow resp.), and j is its position in the output flow. There exists a set of firing relations of the form $\curvearrowright \subset \mathbb{N}^* \times \mathbb{N}^*$, such that for each node (place resp.), $i \curvearrowright j$.

We deduce the following relations from Definitions 11 and 12:

$$\forall p \in \mathcal{P}, \ \forall i \in \mathbb{N}^*, \ i \curvearrowright_p (i + M_0(p)) \tag{2.10}$$

$$\forall\, n \in \mathcal{N}_x \cup \mathcal{N}_c,\ \forall\, i \in \mathbb{N}^*,\ i \curvearrowright_n i \tag{2.11}$$

$$\forall\, s \in \mathcal{N}_s,\ \forall\, i \in \mathbb{N}^*,\ \left[\overline{R(s)}\right]_i \curvearrowright_{s,0} i \tag{2.12}$$

$$\Leftrightarrow\ i \curvearrowright_{s,0} \left|\mathrm{pre}\left(\overline{R(s)}, i\right)\right|_1 \tag{2.13}$$

$$\forall\, s \in \mathcal{N}_s,\ \forall\, i \in \mathbb{N}^*,\ [R(s)]_i \curvearrowright_{s,1} i \tag{2.14}$$

$$\Leftrightarrow\ i \curvearrowright_{s,1} |\mathrm{pre}(R(s), i)|_1 \tag{2.15}$$

$$\forall\, f \in \mathcal{N}_f,\ \forall\, i \in \mathbb{N}^*,\ i \curvearrowright_{f,0} \left[\overline{R(f)}\right]_i \tag{2.16}$$

$$\forall\, f \in \mathcal{N}_f, \forall\, i \in \mathbb{N}^*,\ i \curvearrowright_{f,1} [R(f)]_i \tag{2.17}$$

These relations are monotone: $\forall\ (i_1, j_1), (i_2, j_2) \in \curvearrowright,\ i_1 \leq i_2 \Leftrightarrow j_1 \leq j_2$. Places, copy and computation nodes do not alter token order; there is a bijection between an input and output token index. Relations of merge and select nodes depend on their routing patterns: a select relation is surjective, while a merge relation is injective.

For a given KRG, the set of constraints can be modeled as a dependency graph:

Definition 13 (Dependency graph). A dependency graph (or hypergraph) is a triple $\langle \mathcal{J}, \mathcal{D}, \mathcal{I} \rangle$ where:

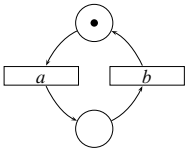
- $\mathcal{J}$ is a finite set of tokens, union of the tokens of the KRG flows.
- $\mathcal{D}$ is a finite multiset of dependencies, each one in $\mathcal{J} \times \mathcal{J}$, and corresponding to a data dependency.
- $\mathcal{I} \subseteq \mathcal{J}$ is the subset of initial tokens.

Dependencies are of three kinds:

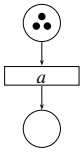
Flow dependency. Basically, a flow dependency between two instructions A and B means that A writes a data in a place, then read by B. In a KRG, this dependency from a token a_1 to a token b_1 means that producing b_1 requires consuming a_1. Figure 2.12a gives an example of cyclic flow dependency. Token b_1 is an initial data. It is consumed by node a to produce a_1, which is then consumed to get back to the initial state. The associated dependency graph is given in Fig. 2.12b.

Output dependency. An output dependency corresponds to consecutive writings into a shared resource. In particular, places behave as FIFOs, and FIFO heads can be seen as such shared resources, as depicted in Fig. 2.12b: a token can be consumed if and only if all its predecessors have already been consumed. The corresponding dependency graph is given in Fig. 2.12e.

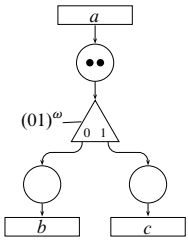
Antidependency. An antidependency between two instructions A and B means that A reads a data in a shared resource before B overwrites it. In our case, it means that an output dependency $a_i \rightarrow a_{i+1}$ and a flow dependency $a_i \rightarrow b_j$ introduce an antidependency $b_j \rightarrow a_{i+1}$: producing b_j (hence consuming a_i) allows to produce a_{i+1}. An example is given in Figs. 2.12c and 2.12f.



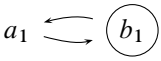
(a) KRG with flow dependency



(b) KRG with output dependency

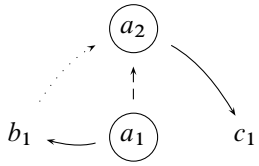


(c) KRG with antidependency



(d) Dependency graph of (**a**). Dependencies between flows a and b

(e) Dependency graph of (**b**). Output dependency in flow a



(f) Dependency graph of (**c**). Antidependency between a_2 and b_1

Fig. 2.12 Examples of the different dependency types

Rule 1 (Constructing dependency graph) *Let $g = \langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0, R \rangle$ be a bounded KRG. We construct its dependency graph $\delta = \langle \mathcal{J}, \mathcal{D}, \mathcal{I} \rangle$ as follows:*

- *We abstract the KRG as a SDF graph and solve balance equations with refined constraints: for each merge node n, we assert that it shall be fired at least $\sum_{p \in \bullet n} M_0(p)$ times over a period. Any other node n' shall be fired at least $\max_{p \in \bullet n} M_0(p)$ times over a period. This way, we compute the number of tokens that we have to consider over a period for each flow.*
- *We associate an unique token $j \in \mathcal{J}$ to each token in each flow of g. A token belongs to $\mathcal{I}$ if it is initially present in its flow (or place).*
- *Tokens of the dependency graph are linked together as mentioned above, according to KRG topology and routing, and with respect to relations of Definition 11. Flow dependencies link pairs of tokens, if the consumption of the first one allows the production of the other one through node firing. For each flow a, an output dependency link each token to its successor: $a_i \rightarrow a_{i+1}$. If, for three tokens a_i, a_{i+1} and b_j, there is a flow dependency $a_i \rightarrow b_j$ and an output dependency $a_i \rightarrow a_{i+1}$, then we create an antidependency $b_j \rightarrow a_{i+1}$.*

The dependency graph is then transposed to an equivalent MG, as stated in this next rule.

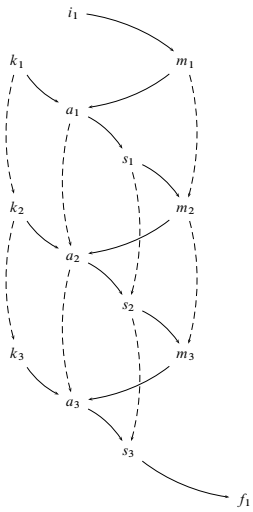
Rule 2 (Constructing equivalent Marked Graph) *A MG* $\langle \mathcal{N}, \mathcal{P}, \mathcal{T}, M_0 \rangle$ *models a dependency graph* $\langle \mathcal{J}, \mathcal{D}, \mathcal{I} \rangle$ *as follows:*

- *Each token* $j \in \mathcal{J}$ *corresponds to a unique quintuplet* $(n, n', p, (n, p), (p, n')) \in \mathcal{N} \times \mathcal{N} \times \mathcal{P} \times \mathcal{T} \times \mathcal{T}$.
- *Let* j_1 *and* j_2 *be two tokens in* $\mathcal{J}$, *and* $\left(n_1, n'_1, p_1, (n_1, p_1), \left(p_1, n'_1\right)\right)$ *and* $\left(n_2, n'_2, p_2, (n_2, p_2), \left(p_2, n'_2\right)\right)$ *their corresponding quintuplets, respectively. If* $(j_1, j_2) \in \mathcal{D}$ *is a flow or output dependency, then* $n'_1 = n_2$, *otherwise* $n'_1 \neq n_2$.
- *For all* $j \in \mathcal{J}$ *and its place* $p \in \mathcal{P}$, $M_0(p) = 1$ *if* $j \in \mathcal{I}$, 0 *otherwise.*

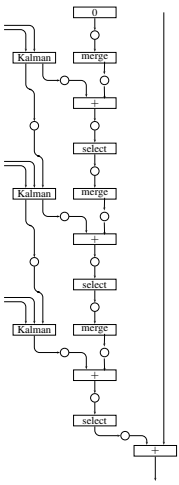
Using this equivalent MG, we can show, as in bounded length simulation, that a KRG is dead-lock free: using any minimum cycle mean algorithm, if the throughput of the MG is greater than 0 then the KRG is deadlock free, otherwise it suffers from deadlocks.

Example 9. We modify slightly the KRG in Fig. 2.10a, considering only three iterations. The routing sequences of select and merge nodes are $\left(0^2.1\right)^\omega$ and $\left(1.0^2\right)^\omega$ respectively.

Using Rule 1, we compute that select, merge and add nodes are executed three times, and the other nodes are fired once over a period. We obtain the dependency graph in Fig. 2.13a. Then, applying Rule 2, we build the equivalent MG, shown in Fig. 2.13b. Multi-dependencies of a given predecessor have been simplified for clarity; it could be further simplified, removing useless select and merge nodes. This graph does not have circuits with empty token count, its throughput is not null and the original KRG is deadlock-free.



(a) Flow and output dependencies



(b) Corresponding MG. Multidependencies have been simplified for clarity

Fig. 2.13 Dependency analysis of KRG in Fig. 2.10a considering three iterations only

2.3.2.5 Topological Transformations

As stated previously, one of the goals of KRG is to enable to build and check correctness of topological transformations of a KRG. Here, we provide simple topological transformations modifying the graph topology of a given KRG while preserving its original semantics: tokens can be routed onto different flows, then merged together, ensuring a compatible order of events with the original topology. In order to check the correctness of the system, we need to introduce operators acting on token flows, in this case on routing patterns attached to select and merge nodes. We recall the *On* operator borrowed from n-synchronous theory [31], and we define the *When* operator.

Definition 14 (*On* operator). The On operator, written $\blacktriangledown$, is recursively defined on binary words as follows: $\forall\, n \in \mathbb{N},\ \forall\, u \in \mathbb{B}^n,\ \forall\, v \in \mathbb{B}^{|u|_1}$,

$$\varepsilon \blacktriangledown \varepsilon = \varepsilon \tag{2.18}$$

$$u \blacktriangledown v, = \begin{cases} 0.\,(u_{\text{tail}} \blacktriangledown v) & \text{if } u_{\text{head}} = 0, \\ v_{\text{head}}.\,(u_{\text{tail}} \blacktriangledown v_{\text{tail}}) & \text{if } u_{\text{head}} = 1. \end{cases} \tag{2.19}$$

Definition 15 (*When* operator). The When operator, written $\triangle$, is recursively defined on binary words, such that: $\forall\, n \in \mathbb{N},\ \forall\, u, v \in \mathbb{B}^n$,
if $n = 0$,

$$u \triangle v = \varepsilon \triangle \varepsilon = \varepsilon \tag{2.20}$$

otherwise,

$$u \triangle v = \begin{cases} u_{\text{queue}} \triangle v_{\text{queue}} & \text{if } v_{\text{head}} = 0, \\ u_{\text{head}}.\,\left(u_{\text{queue}} \triangle v_{\text{queue}}\right) & \text{if } v_{\text{head}} = 1. \end{cases} \tag{2.21}$$

Equivalently: $\forall\, n \in \mathbb{N},\ \forall\, u, v \in \mathbb{B}^n,\ \exists\, w \in \mathbb{B}^{|v|_1}$,

$$u \triangle v = w \ \Leftrightarrow\ \forall\, i \in [\![1, |w|]\!],\ w_i = u_{[v]_i}. \tag{2.22}$$

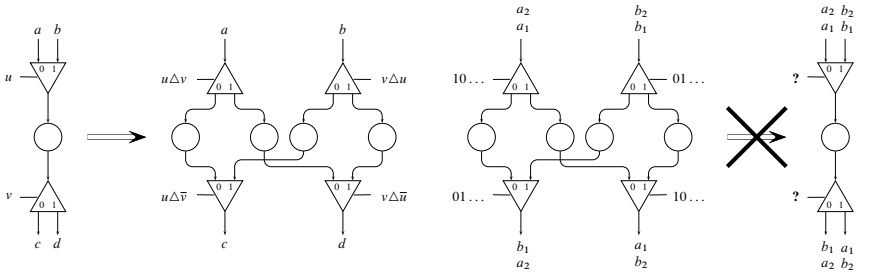
Places can be shared by different token flows: for instance if data are serialized through a shared communication medium. Expanding such places, as shown in Fig. 2.14a, is equivalent to replacing the shared medium by as many point-to-point links as required. This transformation may introduce more concurrency in the KRG but still preserves token order over each flow.

Now, we introduce a useful lemma that simplifies next proof of the expansion of a place. This lemma states an equivalence of flow.

Lemma 2. $\forall\, u, v \in \mathbb{P}, \forall\, i \in \mathbb{N}^*$,

$$v_{[u]_i} = 1 \ \Rightarrow\ |\text{pre}\,(v, [u]_i)|_1 = [u \triangle v]_{|\text{pre}(v \triangle u, i)|_1}\,. \tag{2.23}$$

Proof. We illustrate the proof by the example of Fig. 2.14a (left). One the left side of the implication asserts that the $[u]_i$th token going through the merge node is



(a) Place expansion

(b) Example of graph where reverse factorization is impossible

Fig. 2.14 Expanding and factorizing places

routed onto the first output of the select node, i.e., the ith token in flow b is routed towards flow d. We show that both operands of the equality are different manners to write the index in flow d of this ith token of flow b.

On the left-hand side: $[u]_i$ is the position, in the merge output flow, of the ith token in b (position of the ith occurrence of "1" in the routing sequence). pre $(v, [u]_i)$ corresponds to the routing by select node of tokens, up to the ith from b. We have assumed that the ith letter of the select sequence is "1", so its position in d equals the number of "1" in this prefix.

On the right-hand side: $v \triangle u$ is a sampling by the routing sequence of the select node by the one of the merge node; it corresponds to the routing, by the select node, of all tokens issued from b. Then, $|\text{pre}\,(v \triangle u, i)|_1$ is the number of those tokens, up to the ith one, routed to d. Conversely, $u \triangle v$ corresponds to origins of tokens in d. Finally, $[u \triangle v]_{|\text{pre}(v \triangle u, i)|_1}$ is the position, among tokens in d, of the ith coming from b.

Definition 16 (Order relation on a path). Let Σ be the set of elementary paths in a KRG. A path $\sigma \in \Sigma$ is of the form $\sigma = n_1.p_1.n_2. \ldots .n_{l+1}$, with $n_1, \ldots, n_{l+1} \in \mathcal{N}$ et $p_1, \ldots, p_l \in \mathcal{P}$. We associate a relation $\curvearrowright_\sigma \subseteq (\mathbb{N}^* \times \mathbb{N}^*)$ to each path σ such that $\curvearrowright_\sigma = \curvearrowright_{n_{l+1}} \circ \curvearrowright_{p_l} \circ \ldots \circ \curvearrowright_{n_1}$, where $\curvearrowright_b \circ \curvearrowright_a = \{(i,k) \mid \exists\, (i,j) \in \curvearrowright_a, (j,k) \in \curvearrowright_b\}$.

$\curvearrowright_\sigma$ is a monotone relation since it is composed of monotone relations.

Definition 17 (Order preservation). A KRG is said to be *order-preserving* if and only if $\forall\, n_1, n_2 \in \mathcal{N}$, $\forall\, \sigma_1, \sigma_2 \in \Sigma_{n_1 \rightsquigarrow n_2}$ such that $\sigma_1 \neq \sigma_2$, $\curvearrowright_{\sigma_1} \cup \curvearrowright_{\sigma_2}$ is monotone.

Proposition 1 (Expanding a place). *Expanding a place, such as described in Fig. 2.14a, preserves order relations over token flows.*

Proof. Using and composing flow relations, we infer the following relations between input and output flows:

Figure 2.14a (left):	Figure 2.14a (right):
$a \curvearrowright_{s,0} \circ \curvearrowright_p \circ \curvearrowright_{f,0} \lvert\mathrm{pre}\,(\overline{v}, [\overline{u}]_a)\rvert_1$	$a \curvearrowright_{f,0} \circ \curvearrowright_p \circ \curvearrowright_{s,0} [\overline{u} \triangle \overline{v}]_{\lvert\mathrm{pre}(\overline{v}\triangle\overline{u},a)\rvert_1}$
$a \curvearrowright_{s,1} \circ \curvearrowright_p \circ \curvearrowright_{f,0} \lvert\mathrm{pre}\,(v, [\overline{u}]_a)\rvert_1$	$a \curvearrowright_{f,0} \circ \curvearrowright_p \circ \curvearrowright_{s,1} [\overline{u} \triangle v]_{\lvert\mathrm{pre}(v\triangle\overline{u},a)\rvert_1}$
$b \curvearrowright_{s,0} \circ \curvearrowright_p \circ \curvearrowright_{f,1} \lvert\mathrm{pre}\,(\overline{v}, [u]_b)\rvert_1$	$b \curvearrowright_{f,1} \circ \curvearrowright_p \circ \curvearrowright_{s,0} [u \triangle \overline{v}]_{\lvert\mathrm{pre}(\overline{v}\triangle u,b)\rvert_1}$
$b \curvearrowright_{s,1} \circ \curvearrowright_p \circ \curvearrowright_{f,1} \lvert\mathrm{pre}\,(v, [u]_b)\rvert_1$	$b \curvearrowright_{f,1} \circ \curvearrowright_p \circ \curvearrowright_{s,1} [u \triangle v]_{\lvert\mathrm{pre}(v\triangle u,b)\rvert_1}$

Then, equalities between left and right relations are shown by direct application of Lemma 2.

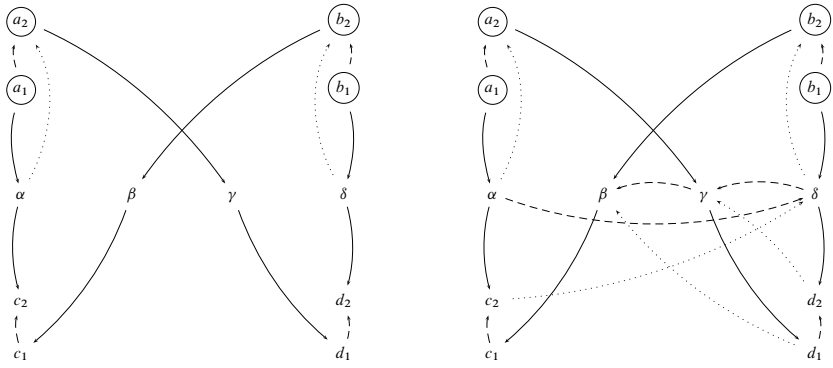
Notice that factorizing places is not always possible, as illustrated in Fig. 2.14b. This is because some token orders allowed by point-to-point connections are incompatible with token sequentialization that generates a total order.

Figure 2.15a presents token dependencies of the example of point-to-point connections in Fig. 2.14b. α, β, γ and δ are names given to tokens in middle places, between select and merge nodes. This graph is acyclic: we can easily find routing patterns of select and merge nodes, going up dependency paths.

In Fig. 2.15b tokens have been arbitrarily sequentialized, as in Fig. 2.14b (right). The middle flow equals $\alpha\delta\gamma\beta = a_1 b_1 a_2 b_2$. In that case, the associated dependency graph is cyclic:

- Circuit 1: $d_2 \leftarrow d_1 \leftarrow \gamma \leftarrow d_2$
- Circuit 2: $c_2 \leftarrow c_1 \leftarrow \beta \leftarrow \gamma \leftarrow \delta \leftarrow c_2$
- Circuit 3: $c_2 \leftarrow c_1 \leftarrow \beta \leftarrow b_2 \leftarrow b_1 \leftarrow \delta \leftarrow c_2$

Circuits 1 and 2 do not have any initial token: there is a deadlock. These circuits can be broken if we permute γ and δ on one side, α and β on the other. Corresponding solutions are as follows: $a_2 b_1 b_2 a_1$, $a_2 b_2 a_1 b_1$, $a_2 b_2 b_1 a_1$, $b_2 a_1 a_2 b_1$, $b_2 a_2 a_1 b_1$



(a) Dependency graph of the expanded form, in Fig. 2.14b (*left*)

(b) Dependency graph of an arbitrarily factorized form, in Fig. 2.14b (*right*)

Fig. 2.15 Comparison of dependencies between expanded and factorized forms: sequentializing introduces dependencies

and $b_2a_2b_1a_1$. None of this word is in $a \amalg b$: such solutions falsify Definition 11. It is impossible to find a valid sequentialization.

Proposition 2 (Merge permutation). *Permuting Merge nodes, as shown in Fig. 2.16a, preserves token orders.*

Proof. From Definitions 12 and 16, we deduce the following relations:

Figure 2.16a (left):	Figure 2.16a (right):
$a \curvearrowright_{f,0} [\overline{v}]_a$	$a \curvearrowright_{f,0} \circ \curvearrowright_p \circ \curvearrowright_{f,0} \left[\overline{v \blacktriangledown u}\right]_{\left[v \triangle \overline{v \blacktriangledown u}\right]_a}$
$b \curvearrowright_{f,1} \circ \curvearrowright_p \circ \curvearrowright_{f,0} [v]_{[\overline{u}]_b}$	$b \curvearrowright_{f,0} \circ \curvearrowright_p \circ \curvearrowright_{f,1} \left[\overline{v \blacktriangledown u}\right]_{\left[v \triangle \overline{v \blacktriangledown u}\right]_b}$
$c \curvearrowright_{f,1} \circ \curvearrowright_p \circ \curvearrowright_{f,1} [v]_{[u]_c}$	$c \curvearrowright_{f,1} [v \blacktriangledown u]_c$

Then, we have:

(a)

$$\begin{aligned}\overline{v \blacktriangledown u} \blacktriangledown \overline{v \triangle \overline{v \blacktriangledown u}} &= (v \blacktriangledown u) \oplus \overline{\overline{v \blacktriangledown u} \blacktriangledown \left(v \triangle \overline{v \blacktriangledown u}\right)} = (v \blacktriangledown u) \oplus \overline{v \wedge \overline{v \blacktriangledown u}} \\ &= (v \blacktriangledown u) \oplus (\overline{v} \vee (v \blacktriangledown u)) = \overline{v}.\end{aligned}$$

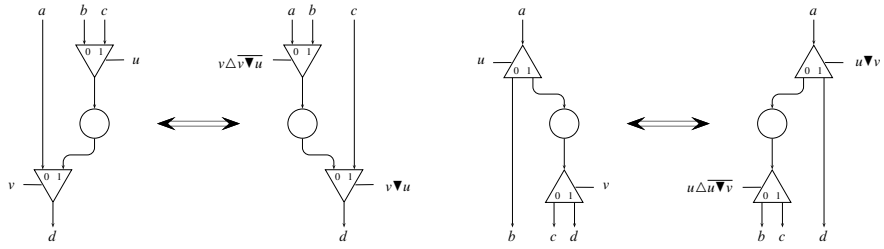
(b)

$$\overline{v \blacktriangledown u} \blacktriangledown \left(v \triangle \overline{v \blacktriangledown u}\right) = v \wedge \overline{v \blacktriangledown u} = v \wedge (\overline{v} \oplus (v \blacktriangledown \overline{u})) = v \blacktriangledown \overline{u}.$$

(c)

$$v \blacktriangledown u = v \blacktriangledown u.$$

Missing algebraic properties and detailed proofs can be found in Boucaron et al. [12].



(a) Permuting merge nodes **(b)** Permuting select nodes

Fig. 2.16 Permuting routing nodes

Proposition 3 (Select permutation). *Permuting Select nodes, as shown in Fig. 2.16b, preserves token orders.*

Proof.

Figure 2.16b (left):	Figure 2.16b (right):
$[\overline{u}]_b \curvearrowright_{s,0} b$	$\left[\overline{u \blacktriangledown v}\right]_{\left[u \triangle \overline{u \blacktriangledown v}\right]_b} \curvearrowright_{s,0} \circ \curvearrowright_p \circ \curvearrowright_{s,0} b$
$[u]_{[\overline{v}]_c} \curvearrowright_{s,0} \circ \curvearrowright_p \circ \curvearrowright_{s,1} c$	$\left[\overline{u \blacktriangledown v}\right]_{\left[u \triangle \overline{u \blacktriangledown v}\right]_c} \curvearrowright_{s,1} \circ \curvearrowright_p \circ \curvearrowright_{s,0} c$
$[u]_{[v]_d} \curvearrowright_{s,1} \circ \curvearrowright_p \circ \curvearrowright_{s,1} d$	$[u \blacktriangledown v]_d \curvearrowright_{s,1} d$

The proof is similar to the one of Proposition 2, and can be found in Boucaron et al. [12].

Proposition 4 (Transformation consistency). *All following transformations: Place expansion (Fig. 2.14a), Merge node permutation (Fig. 2.16a), and Select node permutation (Fig. 2.16b) preserves token flows relations between inputs and outputs. They do not alter neither graph boundedness, nor liveness and deadlock-freedom properties.*

Proof. Using Propositions 1, 2 and 3, these transformations preserve relations between input and output token flows.

Summary

This section provides an overview of static controlled MoCCs. CSDF and KRG have the following properties:

- Deterministic behavior with respect to concurrency
- Decidability on bounded buffers during execution through an abstraction to SDF and solving balance equations
- Dynamic communication topology, that enables reuse of resources for communication, computation and buffers
- Static schedulability: the schedule of a bounded system can be computed at compilation time with needed buffer sizes
- Deadlock-freeness checks using bounded length symbolic simulation
- Correct-by-construction transformations of the communication topology and associated routing patterns in case of KRG

2.4 Conclusion

Modern hardware designs are multicore and even becoming manycore, with possibly heterogeneous processors and direct hardware accelerators. This increased parallelism must match the needs of applications, themselves becoming more and more concurrent in the embedded market, which becomes almost everything in these convergence era. A profusion of formalisms have been proposed to cope with this problem of efficient compilation mapping. We feel that process networks, as part of more general Concurrency Theory, constitute an ideal mathematical field in which to assess the corresponding issues and analyze many of these proposals. One can then recall the impact of formal language theory (automata, Turing machines) on previous sequential computer architectures. In order to fulfill that role, process networks have themselves to be thoroughly characterized and their mathematical properties explicitly described. There is already a large body of results, we try to advance on the topics of scheduling and routing in this context of process networks, and the explicit representations of actual schedules and routes obtained in favorable cases: when static ultimately periodic solutions exist, which is the analogous of finite automata for sequential computations.

This chapter describes some process networks, also called Models of Computations and Communications (MoCCs). Presented MoCCs are simple and natural abstractions for concurrent applications. The simplicity of such MoCCs allows a lot of automation using proved algorithms for compilation/synthesis, performance analysis and formal verification tools. This can enable a better productivity and enable design reuse, it can enable the production of highly-reliable or formally proved products.

We present briefly *pure* dataflow MoCCs such as Synchrony, Latency-Insensitive Design, Marked Graphs and Synchronous Data Flow. All those pure dataflow MoCCs have a deterministic behavior with respect to concurrency and a static communication topology. They generate partial order of events due to conflict-freeness property: this allows a lot of opportunities for scheduling; and this eases to show correctness of an applied transformation. All of them are statically schedulable: performance metrics can be derived at compilation time for both the size of memory resources and the throughput of the system.

After, we present some *statically* controlled MoCCs such as Cyclo-Static Data Flow and K-periodically Routed Graphs. Statically controlled MoCCs allow to *route* data, to change dynamically the communication topology during system execution, but all routing decisions have to be known at compilation time. Statically controlled MoCCS are still ensuring desirable properties such as conflict-freeness and determinism with respect to concurrency. We can check if buffers are bounded during run-time and if so, we can also build a static schedule.

Of course, all previous features come at a cost: all presented MoCCs do not have the same expressiveness as Turing-machines and have a statically known call graph at compilation time.

Nevertheless, the goal of such MoCCs is not to describe all applications, but their expressivities are powerful enough to describe a wide range of useful applications

needed in signal processing (digital filters for audio, image, video), video games (physics, path finding, character animation), genetic research (DNA sequence alignment), biochemistry (molecular dynamics).

Much remains to be done. Many features useful in practice will defeat conflict-freeness property (priority preemption, synchronous real-time) and should be handled with formality still. Conversely, unpredictability due to the many middleware layers or the interconnect fabric and memory access uncertain latencies should also be considered. But while formal techniques may not yet be able to tackle these, at least they make it possible to name them and not try to get rid of the topic falsely by ignoring it. Also, there is tremendous need for MoCC friendly architectures [58] to simplify development of tools: for Worst Case Execution Timing used for Real-Time systems, for highly-optimizing synthesizers/compilers and for formal verification tools.

References

1. S. Amarasinghe. A stream compiler for communication-exposed architectures. In *Proceedings of ASPLOS*, 2002.
2. F. Anceau. A synchronous approach for clocking VLSI systems. *IEEE Journal of Solid-State Circuits*, 17:51–56, 1982.
3. C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *CESA*, 1996.
4. F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*. Wiley, Chichester, West Sussex, UK, 1992.
5. D. Baneres, J. Cortadella, and M. Kishinevsky. Variable-latency design using function speculation. In *Proc. Design, Automation and Test in Europe*, April 2009.
6. L. Benini, E. Macii, and M. Poncino. Telescopic units: Increasing the average throughput of pipelined designs by adaptive latency control. In *DAC*, pages 22–27, 1997.
7. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
8. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, Dordrecht, 1996.
9. G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44:397–408, February 1996.
10. A. Bouali. XEVE, an esterel verification environment. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 500–504, London, UK, 1998. Springer.
11. J. Boucaron, A. Coadou, and R. de Simone. Latency-insensitive design: retry relay-station and fusion shell. In *Formal Methods for Globally Asynchronous Locally Synchronous Design 2009 Proceedings*, 2009.
12. J. Boucaron, A. Coadou, B. Ferrero, J.-V. Millo, and R. de Simone. Kahn-extended event graphs. Research Report RR-6541, INRIA, 2008.
13. J. Boucaron, J.-V. Millo, and R. de Simone. Another glance at relay-stations in latency-insensitive design. In *Formal Methods for Globally Asynchronous Locally Synchronous Design 2005 Proceedings*, 2005.

14. J. Boucaron, J.-V. Millo, and R. de Simone. Latency-insensitive design and central repetitive scheduling. In *Proceedings of the 4th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE'06)*, pages 175–183, Napa Valley, CA, USA, July 2006. IEEE Press.
15. J. Boucaron, J.-V. Millo, and R. de Simone. Formal methods for scheduling of latency-insensitive designs. *EURASIP Journal on Embedded Systems*, 2007(1), 8–8, 2007.
16. R. G. Brown and P. Y. C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*, 3rd edition. Wiley, New York, 1996.
17. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 1986.
18. J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. PhD thesis, University of California, Berkeley, CA, USA, 1993.
19. D. Bufistov, J. Júlvez, and J. Cortadella. Performance optimization of elastic systems using buffer resizing and buffer insertion. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 442–448, Piscataway, NJ, USA, 2008. IEEE Press.
20. J. L. Campbell. *Application of Airborne Laser Scanner – Aerial Navigation*. PhD thesis, Russ College of Engineering and Technology, Athens, OH, USA, February 2006.
21. L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A methodology for correct-by-construction latency-insensitive design. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'99)*, pages 309–315, Piscataway, NJ, USA, November 1999. IEEE.
22. L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.
23. L. P. Carloni and A. L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 361–367, New York, NY, USA, 2000. ACM.
24. L. P. Carloni and A. L. Sangiovanni-Vincentelli. Combining retiming and recycling to optimize the performance of synchronous circuits. In *SBCCI '03: Proceedings of the 16th symposium on Integrated circuits and systems design*, page 47, Washington, DC, USA, 2003. IEEE Computer Society.
25. P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.
26. P. Caspi and M. Pouzet. A functional extension to Lustre. In M. A. Orgun and E. A. Ashcroft, editors, *International Symposium on Languages for Intentional Programming*, Sydney, Australia, May 1995. World Scientific.
27. P. Caspi and M. Pouzet. *Lucid Synchrone, version 1.01. Tutorial and reference manual*. Laboratoire d'Informatique de Paris 6, January 1999.
28. M. R. Casu and L. Macchiarulo. A detailed implementation of latency insensitive protocols. In *Proceedings of the 1st Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS'03)*, pages 94–103, September 2003.
29. M. R. Casu and L. Macchiarulo. A new approach to latency insensitive design. In *Proceedings of the 41st Annual Conference on Design Automation (DAC'04)*, pages 576–581, 2004.
30. M. R. Casu and L. Macchiarulo. Adaptive latency-insensitive protocols. *IEEE Design and Test of Computers*, 24(5):442–452, 2007.
31. A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 180–193, New York, NY, USA, January 2006. ACM.
32. F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graph. *Journal of Computer and System Sciences*, 5:511–523, October 1971.
33. J. Cortadella, M. Kishinevsky, and B. Grundmann. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Annual Conference on Design Automation (DAC'06)*, pages 657–662, New York, NY, USA, 2006. ACM.

34. M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini. Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SOCS. In *ICCD*, pages 536–, 2003.
35. A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, Boston, 2000.
36. A. Dasdan and R. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.
37. G. de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York, 1994.
38. N. Eén and N. Sörensson. An extensible SAT solver. In *SAT Proceedings*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, Berlin, 2003.
39. J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
40. M. Engels, G. Bilsen, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow: model and implementation. In *Conference Record of the Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 503–507, Pacific Grove, CA, USA, 1994.
41. D. Gebhardt and K. Stevens. Elastic flow in an application specific network-on-chip. In *Formal Methods for Globally Asynchronous Locally Synchronous Design Proceedings*, 2007.
42. R. Govindarajan and G. R. Gao. Rate-optimal schedule for multi-rate DSP computations. *Journal of VLSI Signal Processing*, 9(3):211–232, 1995.
43. T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.
44. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
45. G. Hoover and F. Brewer. Synthesizing synchronous elastic flow networks. In *DATE*, pages 306–311, 2008.
46. D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.
47. G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475, New York, NY, USA, 1974. North-Holland.
48. M. Karczmarek. Phased scheduling of stream programs. In *Proceedings of LCTES*, 2003.
49. R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.
50. R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, May, 1969.
51. K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, San Francisco, CA, USA, 2002.
52. M. Kudlur and S. A. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI*, pages 114–124, 2008.
53. P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. *Journal for Circuits, Systems and Computers*, 12:261–304, 2002.
54. C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38:985–999, 1959.
55. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, 1987.
56. E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, 75(9):1235–1245, 1987.
57. C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.
58. B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*, Atlanta, Georgia, USA, October 19–24, 2008.

59. N. Liveris, C. Lin, J. Wang, H. Zhou, and P. Banerjee. Retiming for synchronous data flow graphs. In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 480–485, Washington, DC, USA, 2007. IEEE Computer Society.
60. O. Marchetti and A. Munier-Kordon. Minimizing place capacities of weighted event graphs for enforcing liveness. *Discrete Event Dynamic Systems*, 18(1):91–109, 2008.
61. O. Marchetti and A. Munier-Kordon. A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2):532–540, 2009.
62. K. L. McMillan. *Symbolic Model Checking*. Kluwer, Dordrecht, 1993.
63. T. O'Neil and E.-M. Sha. Retiming synchronous data-flow graphs to reduce execution time. *IEEE Transactions on Signal Processing*, 49(10):2397–2407, 2001.
64. H. D. Patel and S. K. Shukla. *SystemC Kernel Extensions For Heterogenous System Modeling: A Framework for Multi-MoC Modeling and Simulation*. Kluwer, Norwell, MA, USA, 2004.
65. C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, Germany, 1962.
66. D. Potop-Butucaru, S. A. Edwards, and G. Berry. *Compiling Esterel*. Springer, Berlin, 2007.
67. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, 1982.
68. C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA, USA, 1973.
69. M. D. Riedel. *Cyclic Combinational Circuits*. PhD thesis, Dept. of Electrical Engineering, California Institute of Technology, Pasadena, CA, USA, November 2003.
70. K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.
71. J. Sermulins. Cache aware optimization of stream programs. In *Proceedings of LCTES*, 2005.
72. C. Soviani, O. Tardieu, and S. A. Edwards. High-level optimization by combining retiming and shannon decomposition. In *In Proceedings of the International Workshop on Logic and Synthesis (IWLS)*, Lake Arrowhead, CA, June, 2005.
73. S. Suhaib, D. Mathaikutty, D. Berner, and S. K. Shukla. Validating families of latency insensitive protocols. *IEEE Transactions on Computers*, 55(11):1391–1401, 2006.
74. S. Suhaib, D. Mathaikutty, S. K. Shukla, D. Berner, and J.-P. Talpin. A functional programming framework for latency insensitive protocol validation. *Electronic Notes in Theoretical Computer Science*, 146(2):169–188, 2006.
75. C. Svensson. Synchronous latency insensitive design. In *Proc. of 10th IEEE International Symposium on Asynchronous Circuits and Systems*, 2004.
76. W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Lin, A. Meli, A. Lamb, C. Leger, and S. Amarasinghe. Streamit: a language for streaming applications. In *Proceedings of New England Programming Languages and Systems Symposium (NEPLS)*, 2002.

http://www.springer.com/978-1-4419-6399-4

Synthesis of Embedded Software
Frameworks and Methodologies for Correctness by Construction
Shukla, S.K.; Talpin, J.-P. (Eds.)
2010, XV, 266 p., Hardcover
ISBN: 978-1-4419-6399-4