Statically scheduled Process Networks

Julien Boucaron, Benoît Ferrero, Jean-Vivien Millo, Robert de Simone

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Statically scheduled Process Networks

Julien Boucaron — Benoît Ferrero — Jean-Vivien Millo — Robert de Simone

N° 6289

Août 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6289--FR+ENG

Statically scheduled Process Networks

Julien Boucaron , Benoît Ferrero , Jean-Vivien Millo , Robert de Simone

Thème COM — Systèmes communicants
Projets Aoste

Rapport de recherche n° 6289 — Août 2007 — 26 pages

Abstract:

Event/Marked Graphs (EG) form a strict subset of Petri Nets. They are fundamental models in Scheduling Theory, mostly because of their absence of alternative behaviors (or *conflict-freeness*).

It was established in the past that, under broad structural conditions, behavior of Timed Event Graphs (TEG) becomes utterly regular (technically speaking: "*ultimately k-periodic*"). More recently it has been proposed to use this kind of regular schedulings as syntactic types for so-called *N-synchronous* processes. These types remained essentially user-provided.

Elsewhere there have been proposals for adding control in a "light fashion" to TEGs, not as general Petri Nets, but with the addition of *Merge/Select* nodes switching the data flows. This was much in the spirit of Kahn process networks [8, 9]. But usually the streams of test values governing the switches are left unspecified, which may introduce phenomena of congestion or starvation in the system, as token flow preservation becomes an issue.

In the present paper we suggest to restrict the Merge/Select condition streams to (binary) k-periodic patterns as well, and to study their relations with the schedules constructed as before for TEGs, but on the extended model. We call this model *Kahn-extended Event Graphs (KEG)*.

The main result is that flow preservation is now checkable (by abstraction into another model of Weighted Marked Graphs, called SDF in the literature).

There are many potential applications of KEG models, as for instance in modern Systems-on-Chip (SoC) comprising on-Chip networks. Communication links can then be shared, and the model can represent the (regular) activity schedules of the computing as well as the communicating components, after a strict scheduling has been found. They can also be used as a support to help find the solution.

Key-words: repetitive cyclic scheduling, binary, formal models, synchronous

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Statically scheduled Process Networks

Résumé : Les Graphes d'évènement (EG) forment un sous-ensemble strict des Réseaux de Petri. Etant dénués de tout choix non-déterministe ils jouent un rôle fondamental dans la Théorie de l'Ordonnancement.

Il a été établi dans le passé que, sous des hypothèses naturelles sur la forme structurelle des graphes, on savait construire une version temporisée de tels graphes d'evènement (TEG), qui admettait un ordonnancement finalement régulier (en terms techniques: *ultimement k-périodique*). Plus récemment il a été proposé d'utiliser ces formulations d'ordonnancement régulier comme types syntaxiques pour la conception de systèmes dits *N-synchrones*. Ce typage reste essentiellement fourni par l' utilisateur.

Par ailleurs des extensions aux EGs ont aussi été proposées dans le passé, afin d'introduire une "certaine dose" de contrôle (if-then-else), mais sans réintroduire toute la généralité des RdP. ceci repose sur des noeuds de contrôle *merge* et *select*, dont les branchements sont entièrement commandés par des conditions locales, distinctes des apparitions de valeurs aux canaux d'entrée, ce qui est dans la droite tradition des Réseaux de Kahn. Mais ces conditions sont en générale abstraites par cette propriété, ce qui introduit des questions de famine ou de congestion des canaux qui ne peuvent alors être résolues.

Dans ce papier nous introduisons une notion de conditions de branchement k-périodiques sur ces noeuds *Merge/Select*. Le résultat prinipal est que le problème de l'égalisation des flots dans ce modèle redevient décidable, et qu'on peut y redéfinir une notion d'ordonnancement k-périodique qui combine la régularité "en profondeur" des latences avec celle "en largeur" des branchements alternatifs. Nous nommons ce modèle Graphes d'Evènement Kahn-augmentés (KEG).

Il y a de nombreuses applications potentielles à ce m odèle KEG, en particulier dans le design de Systèmes-sur-puce (SoC) comprenant un Réseau-sur-puce (NoC). Dans des cas où les traffics peuvent être prévisibles (statiquement), le modèle permet de représenter l'entrelacement des routages afin de partager optimalement des sections de communication et acheminer les données sur un mode régulier. Il peut aussi permettre de trouver ces ordonnancements de trafic.

Mots-clés : ordonnancement cyclique répétitif, binaire, modèles formels, synchrone

1 Introduction

General concurrent Models of Computation for embedded systems, such as Petri Nets or Process Algebras, are inherently non-deterministic. Determinism is nevertheless a highly desirable feature for such systems, leading to predictable and reproducible behaviors. Non-determinism may originate from two sources, *internal* or *external*. In internal non-determinism the local conditional choices (on local values) are abstracted away as little meaningful. In external non-determinism it is the unknown relative speeds of communications in untimed setting that may lead to signal notifications being received in different orders, the first received possibly disabling the effect of the second.

Various restricted models of computation have been proposed to preserve determinism and concurrency altogether:

In *Event Graphs*, as in any Petri Nets, the local computation nodes (also called "transitions") consume and produce systematically on *all* of their input and output places respectively. Internal non-determinism is removed by requiring that places have at most one input and output transition, so that they act as buffering channels. Then potential external non-determinism becomes harmless, because faster signals just have to await for slower ones to be processed simultaneously. The result is that computations amount to partially-ordered computation traces.

In the *Kahn process network* model the internal non-determinism is strongly controlled and imperatively provided (even though it is usually abstracted as a corresponding property), so that the order in which signals are consumed and produced at individual process components is, here again, independent of their order of arrival. To be more specific in event graphs data/signal "tokens" are consumed and produced "all at once" simultaneously, while in Kahn networks they are consumed and produced individually, but in a way internally prescribed and independent of their availability in the environment (and several data can be consumed on a channel before one is consumed or produced on another, for instance). We shall try in the sequel to combine both operational modes in a specific way.

When provided with an ASAP semantics (each computation node fires as soon as it may), strongly connected Event Graphs reveal an *ultimately k-periodic* scheduled behavior. What this means is that, after an initial stabilization phase, each computation node fires according to a finite repeated pattern relative to the global synchronous time of execution cycles. This now classical result, due to [12, 5, 2], can be taken as a basis to allocate an explicit scheduling pattern to each node, as an ultimately periodic binary word (where "0" means inactivity, and "1" means firing at this precise global instant). Our idea of using explicit schedule objects in embedded systems design methodology goes back to the theory of *N-synchronous* processes [6], where it was used in a "behavioral type" framework to ensure that (user-provided) schedules were consistent so that the system's production rates was well-balanced, and only finite buffers were needed for inclusion in the design to support finite delay variations. The notion is also related to the notion of affine clocks in the Signal and Alpha languages [1].

In [3] we use it to effectively build explicit schedules for computation nodes of Latency-Insensitive Designs, and simplify the requested congestion control protocol resources accordingly (and drastically).

We now want to extend this work of using explicit schedules on ultimately k-periodic networks by applying to a combination of Event Graphs and Kahn networks. For simplicity we restrict the "Kahn-aspect" to specific *Select* and *Merge* nodes, with deterministic and internally prescribed branch condition flows. Of course to adapt our model we will simply request that such branch condition flows be themselves k-periodic (reproducing the same pattern over and over). But this time "0" means "left", or "else", and "1" means "right/then".

We provide extensive model definitions. We show that the *safety* (with the meaning of *safe* as in Petri Nets) property can be reduced to a condition of *balanced equations* in the SDF formalism (itself another extension of Event Graphs). This provides a useful decision criterion for establishing that a "Kahn-extended Event Graph" (KEG) has a finite reachable state space. From this, model-checking based techniques allow to check for *liveness* (or deadlock-freedom), to compute the actual schedules of computation nodes, and even to obtain actual sizes of requested buffering elements needed to store the data/signal values that may be halted waiting for the branching node to authorize their flow to proceed.

As a side result we recall (while it is never explicitly stated to the best of our knowledge) how formalisms of SDF and Timed Event Graphs, with integer latencies on computation nodes or communication channels, can simply be expanded into plain ones that are equivalent wrt ASAP semantics.

We cast our result in the continuation of our works on static, ultimately k-periodic scheduling of Latency-Insensitive Designs, with applications to (mainly data-flow) SoC design integration and timing closure.

Related Works Most closely related to ours is the work of Edward Lee and colleagues, on Synchronous Data Flow graphs (SDF) and extensions [4, 11]. They obtain several results for bounds and decidability (or undecidability) of schedulings, but never use (as far as we know) periodic switching condition flows to regular Kahn networks. Other sources of inspiration were, as mentioned previously, N-synchronous and multiclock synchronous formalisms, and cyclic scheduling theory based on Timed Event Graphs.

Outline In the next Section we introduce Event Graphs, Synchronous Data Flow graphs (SDF), Timed Event Graphs and Scheduling notation borrowed from N-Synchronous theory. After we describe our Kahn-extended Event Graph (KEG) which is a determinist *à la Kahn* sub-class of Petri Net with specific control nodes annotated by N-Synchronous words, where we can show buffer boundedness through a sound abstraction over SDF Net. Finally, we conclude.

2 Background definitions

We shall first require a number of definitions and notations, including Event Graphs, their ASAP firing rules and their Weighted (SDF) and Timed extensions. Then we shall recall k-periodic scheduling result, and the syntax for representing them. It is based on periodic binary words together with operators on them, which we shall also introduce.

2.1 Event Graphs

Definition 1 (Event Graph) *An* Event Graph (EG)*, also called Marked Graph [7] in the literature, is a structure* $E =< \mathcal{C}, \mathcal{P}, M >$*, where:*

- $\mathcal{C}$ *is a finite set of* computation nodes *(the vertices of the graph, the* transitions *of the Petri net);*
- $\mathcal{P} \subset \mathcal{C} \times \mathcal{C}$ *is a finite set of buffer channels (the edges of the graphs, the* places *of the Petri net);*
- $M \in (\mathcal{P} \longrightarrow \mathbb{N})$ *assigns an (initial) token marking to places.*

We note $c = Source(p)$ *and* $c' = Target(p)$ *when* $p = (c, c')$*. Conversely we note* $In(c) = \{p \in \mathcal{P} / \exists c', p = (c, c'\}$ *(and similarly for* $Out(c)$*).*

We slightly departed here from the classical Petri-style definition, to stress the fact that places were here mere channels with exactly one target and one source transition. This is in line with the fact that our latter extensions will *not* go in the direction of more general Petri Nets, but will introduce routing nodes on channels instead.

The dynamic semantics of event graphs is still based on firing steps and firing rules. A transition may fire whenever there is at least one token in *each* of its input places, and as a result produces one token in *each* of its output places. Of course here the good news is that transitions cannot "steal" tokens from one another, as each channel has only one consuming transition. So when a subset of transition are fireable independently, then they can be fired simultaneously.

Definition 2 (Firing step, ASAP firing rule) *Given* $C \subseteq \mathcal{C}$ *a set of computation nodes, we say that* E *fires* C *in a step and becomes* E'*, noted* $E \xrightarrow{C} E'$*, iff:*

- $\forall c \in C, \forall p \in In(c),\ M(p) > 0$
- E' *differs from* E *only by its marking:*
  $M'(p) = M(p) + \delta_{Source(p)} - \delta_{Target(p)}$*, where* $\delta_c = 1$ *iff* $c \in C$*,* $\delta_c = 0$ *otherwise.*

A sequence of firing steps is called fair *iff whenever a computation node is fireable, it will eventually be so. A firing sequence is also called* ASAP *if at each step* i *the set* C_i *of computation nodes fired is maximal (all transitions that may fire do so).*

The asap firing sequence is the "optimal" run of the Event Graph, in which each computation is performed as soon as ready. All other possible fair executions are obtained by delaying some individual transition firings.

Note that if the initial marking puts (at least) one token in each place, then the asap firing executes all computation nodes simultaneously at each step, therefore providing a *synchronous* execution.

Desirable properties on EGs ask that the number of tokens/data are kept between bounds, allowing infinite behaviors to be generated without running dry of data or overflowing place storage abilities.

Definition 3 (Liveness, Safety) *A computation $c \in \mathcal{C}$ is called* live *iff it can always be fired from any reachable marking. A place is called* k-safe *if it never holds more than k tokens in any computation sequence. E is called* live *(resp.* k-safe*) when all its transitions (resp.* places *are. It is called simply safe when it is k-safe for some integer k.*

A strongly connected Event Graph is obviously *safe*, since the number of tokens in each loop remains invariant from this of the initial marking. A strong result by Commoner-Holt-Pnueli [7] (also established independently by Genrich) states that a strongly connected Event Graph is *live* whenever there is at least a token in each graph cycle. But our more abstract formulation extends to larger classes of graphs (such as SDF, see below).

Definition 4 (Throughput) *Given a cycle γ in an Event Graph E, the number of tokens t_γ in γ remains invariant through firings. So the throughput of γ, defined as $length(\gamma)/t_\gamma$ and noted τ_γ, is well defined.*
The throughput of E, noted τ_E, is defined as $min_\gamma\, \tau_\gamma$ (and $\tau_E = \infty$ when there is no cycle in E). Any cycle γ with this minimal throughput is called critical.

We shall also use variants of Event Graphs, defined now. Our namings may vary from the literature, but to our excuse this is mostly because there is no fully established naming that allow to describe all these models distinguishly.

Definition 5 (Bounded Event Graphs) *A* Bounded Event Graph (BEG) *is a regular EG with an additional function $Cap : \mathcal{P} \longrightarrow \mathbb{N}_+$ providing a (positive integer) maximal capacity for the number of tokens the place could hold at most.*
Accordingly the firing rule must be adapted so that firing of a computation node is allowed only if it does not exceeds any of its outgoing places capacity.

Unlike for safety, capacity is a requirement on the system (not a property established). BEGs can be transformed back into plain EGs with the simple addition of one place/channel for each existing one p, except leading on the opposite direction, and holding $Cap(p) - M(p)$ tokens initially. Details are left to the reader.

2.2 Synchronous Data Flow graphs

Definition 6 (Synchronous Data Flow graphs) *A* Synchronous Data Flow *graph* (SDF) *is a regular EG with an additional function $W : \mathcal{P} \longrightarrow (\mathbb{N}_+ \times \mathbb{N}_+)$ providing a couple a*

positive integers, which indicate how many tokens the source and target computation nodes produce and consume respectively from the channel (for plain EGs they are both "1"). Accordingly the firing rule must be adapted so that firing of a computation node is allowed only when there is enough tokens in its input channels.

SDF processes preserve the property that all fair firings sequences are essentially the same as the asap "fastest one", just possibly delayed (the behaviour consists of the same partially-ordered infinite trace).

A general condition known as *Balanced Equations* exists on SDF systems to enforce global flow preservation. Well-balanced SDFs can be expanded into plain EGs, but this time the construction is much more complex.

Definition 7 (Balanced Equations [10]) *"Let G be a SDF graph. Consider the matrix M constructed by assigning a column to each computation node $c \in C$ and a row to each place $p \in P$. Then the $(i,j)^{th}$ entry in M is filled with the amount of data flowing between node index j and place index i each time this node is fired: If node j* produces *tokens to place i this number is positive, it is negative if node j* consumes *tokens from place i; finally it is* 0 *if the place is not directly linked to the node."*
Then G is called well-balanced *(or* respecting the balance equations*) iff $rank(M) = size(\mathcal{C}) - 1$.*

This definition looks somehow involved. What it truly says is that the productions/consumptions of tokens in various parts of the systems have to be proportional, so that there exists constants f_c, one for each computation node c, so that whenever these nodes are triggered each f_c times, then the number of tokens consumed and produced to and from each channel/place is actually equal, so that the system's token flow is, indeed, "well-balanced".

Property 1 *Well-balanced strongly-connected systems are* safe. *Provided there is "enough" tokens in the initial markings, they are also* live.

The proof (see [10]) uses the fact that all three integers attached to a place edge (the two weights and its initial marking) can be proportionally multiplied by the same factor without changing the semantics of the SDF. Then, because of well-balancedness, there exist such multiplicative values such than each computation node becomes *homogeneous*, in the sense that all weights on its side of incoming or outgoing place edges are equal. In the transformed version, the number of tokens in each simple graph cycle remains invariant, which enforces safety and liveness then in a way similar to plain Event Graphs.

Still, figuring out a minimal live marking for SDF graph remains more tricky (some obvious upper bounds exist, but we have counter-examples showing that they are *not* minimal in general).

2.3 Timed Event Graphs

We now provide a fairly broad definitions of Timed Event Graphs, which may impose *latencies* on both computations and communications through channels, and *delays* on computa-

tions. A latency shall measure the time (in number of steps) between the consumption of inputs and the production of outputs by a computation node. A delay shall instead measure the minimal time required between two successive input consumptions. If the computation induces some amount of pipelining, or on the opposite requires internal resets between distinct computations after the result is produced, then latency and delay may differ.

Definition 8 (Timed Event Graphs) *A* Timed Event Graph (TEG) *is a regular EG with two additional functions:* $L : (\mathcal{P} \cup \mathcal{C}) \longrightarrow \mathbb{N}$ *and* $D : (\mathcal{P} \cup \mathcal{C}) \longrightarrow \mathbb{N}_+$*, verifying the following property:*
$\forall \gamma$ *cycle in the graph* E, $\sum_{p\in\gamma} L(p) + \sum_{c\in\gamma} L(C) > 0$ *(no combinatorial, zero-time loops).*

Under the stronger condition that all places in $\mathcal{P}$ have a strictly positive latency, TEGs can be expanded into plain EGs. A sketch of this expansion is provided in figure 1. It relies on two simple ideas: the first one is to introduce *begin/end* computation nodes instead of the former ones, so that computation delays and latencies can be considered as transportation times on the linking edges inside such blocks; then one introduces as many new intermediate transitions as needed (we call them transportation nodes (TN) as opposed to the previous computation nodes (CN)), so that latencies and delays are split down to unitary time lengths.

In the more general case where they are nodes or places with 0-latency figures (instantaneous), the expansion can be performed also, resulting of a mix of only single latency and 0-latency components. The firing rule must then be slightly expanded, so that nodes may be fired in sequel until reaching a fix-point for that step. The transformation is well-defined due to the fact that we still forbid instantaneous loops. For sake of room details are left to the reader.

In fact, the previous expansion is actually *required* to provide a sound a easy formulation of the timed firing rules for TEGs. Indeed, it demands to recall the "age" of each token inside a channel of long latency. With the expansion, tokens will progress by passing successive sections of the channel, each requiring a unitary time step to be crossed.

2.4 Schedules

It can be established that asap firings of a strongly connected EG form an *ultimately k-periodic* pattern for each computation node. We shall introduce notations, most borrowed from [6], to describe such schedules and introduce formally the main results.

Individual schedules will be infinite binary words, with "0" meaning *inactive/idle*, and "1" meaning *fired/executed* at this specific step. Similarly the Merge/Select switch conditions will also be infinite binary words, this time with "0" and "1" meaning *left* and *right* (or `then/else` branches). We know introduce auxiliary notations to deal ith such words as infinite sequences.

Definition 9 (Infinite binary words) *We note as* $\mathbb{B}^{\mathbb{N}}$ *the set of infinite binary words (or Boolean* streams*, or sequences of Booleans).*
For $w \in \mathbb{B}^{\mathbb{N}}$ *we note* $w(i), w(i) \in \mathbb{B}$ *the value of its* i^{th} *position.*
We note $[w]_i$ *the index of its* i^{th} *occurring* $\mathbb{1}$ *(it can be infinite to indicate that there is only*

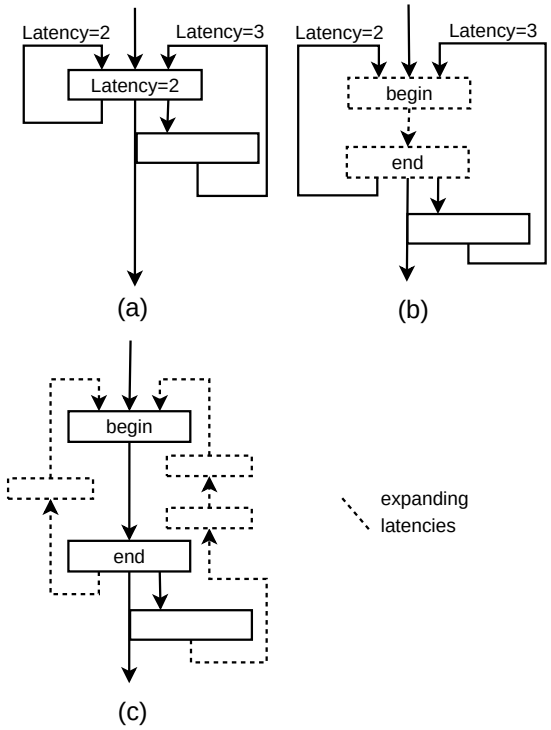

Figure 1: Successive expansions from a Timed Event Graphs (a): splitting computations with `begin/end` transitions whenever computation latency or delay > 1 (b); then splitting edges whenever transportation latency > 1 (c)

a finite number of $\mathbb{1}$ *occurring in* w*, and* $[w]_0 = -1$*).*
We also note $\bar{w}$ *for the complement of* w *(obtained by exchanging all* $\mathbb{0}$*s and* $\mathbb{1}$ *letters in* w*).*
We call w fair *if it contains an infinite number of* $\mathbb{1}$*s and an infinite number of* $\mathbb{0}$*.*
For $w, w' \in \mathbb{B}^{\mathbb{N}}$*, we note* $w \sqsubseteq w'$ *when* $\forall i, (w(i) = \mathbb{1}) \Rightarrow (w'(i) = \mathbb{1})$*, and we then call* w *a* subflow *of* w'*. We also note* $w \prec w'$ *when* $\forall i, [w]_i \leq [w']_i$*.*

The local scheduling patterns and conditions streams of interest will be regular, and more specifically *ultimately periodic*.

Definition 10 (Ultimately periodic words) *We call* ultimately k-periodic word *an infinite word* $w = u.v^{\omega}$*, with* $u, v \in \mathbb{B}^{\star}$*. We let* $\mathcal{KP}$ *be the set of all ultimately k-periodic words.*
In addition we call w immediately k-periodic *iff* $u = \epsilon$ *the empty word.*
We call $|v|$ *(the length of* v*) the* period *of* w*,* $|v|_1$ *(the number of* $\mathbb{1}$ *letters in* v*) the* periodicity *of* w*, and* $|v|_1/|v|$ *the rate of* w *(denoted also* $rate(w)$*).*
Finally we write $w \bowtie w'$ *when* $rate(w) = rate(w')$*.*

Definition 11 (Schedules) *A* schedule *for an event graph* E *is an assignment function* $Sched : \mathcal{C} \longrightarrow \mathbb{B}^{\mathbb{N}}$
The schedule is called admissible *iff*
$\{C_n \,/ C_n = \{c \in \mathcal{C}, Sched(c)(n) = \mathbb{1}\}\}$ *is an allowable firing sequence according to the token distribution (starting from the initial marking).* $Sched$ *is called* asap *iff* $\{C_n\}$ *is an asap firing sequence.*
It is called ultimately k-periodic *iff* $\forall c \in \mathcal{C}$, $Sched(c) \in \mathcal{KP}$*.*

Property 2 (k-periodic asap schedule,[2, 5]) *A strongly connected Event Graph admits an* ultimately k-periodic schedule*.*
In addition all $Sched(c)$ *have the same rate, which is the throughput of the graph (the sum of tokens over the sum of latencies in critical cycles).*
Last, the length of the common period can be computed (technically it is the lcm *over strongly connected critical parts of the* gcd *of all latencies between the various cycle of that given strongly connected critical part).*

The proofs are to be found in the papers of Carlier-Chrétienne, Cohen-Baccelli-Quadrat-Jan Olsder [2, 5]. Even though the exact results on achievable lengths are quite involved, what matters here is that critical cycles "dictate"their pace to the rest of the system, after a while of initial chaos that precedes stabilization.

The previous property can be lifted back to WEGs and TEGs by considering their expansions into plain EGs described before. Figure (c) 2 describes the asap schedule of our example.

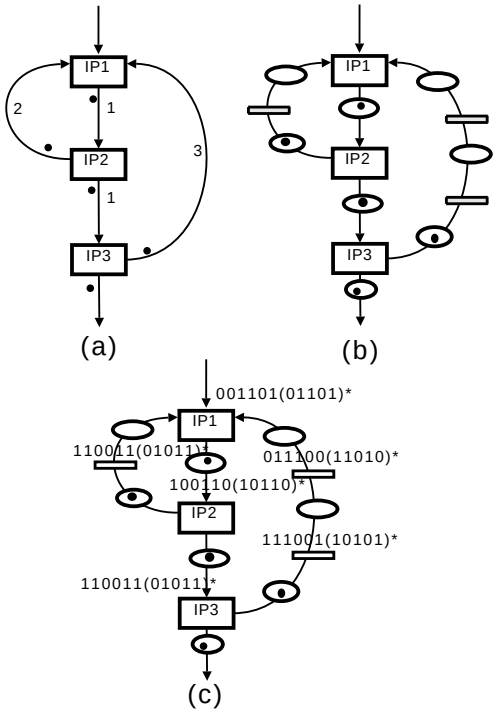

Figure 2: (a) TEG (b) Expanded TEG (c) ASAP Scheduled TEG

3 Adding k-periodic Select/Merge control streams

3.1 Kahn-extended Event Graphs

We shall now introduce new nodes (*Select*s and *Merge*s) to enhance the expressivity of our formalism with some "controlled amount of control".

Before we provide the formal definition and technical apparatus, let us consider the example of figure 3. It represents a (much simplified) multiprocessor interconnect topology in the way of the STI CELL chip. It contain two kind of switches (mux/demux like), both represented by triangles. Merge nodes unite two flows, and select nodes split one flow into two. Two peripheral rings are potentially connecting four C_i processors, in mutually converse directions. Connections depend on how the switch connectors are set (this is also reminiscent of FPGA technologies). Figure 4 shows a switch setting where each processor C_i is connected in full duplex with processor C_{i+2} (one link each way). Conversely in figure 5 the full duplex connection is established between the two first (and two last) processors. The reader should consider to which extend the values of distinct switches must be correlated to form proper closed paths; therefore the topological changes must be carefully scheduled and synchronized. To the best of our knowledge such a topic is not currently addressed in formal modeling terms.

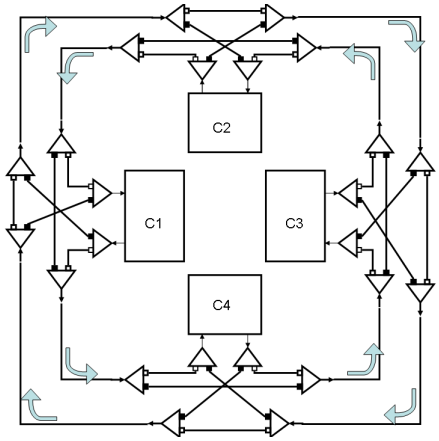

Figure 3: An example KEG: a simplified CELL network

Our goal in this paper shall be to provide expressive mathematical models so that routes can easily be set and reset, or checked for consistency. We also want to be able (under our static scheduling policy) to be able to compute sizes of buffering resources needed for holding data values temporally while routes are modified.

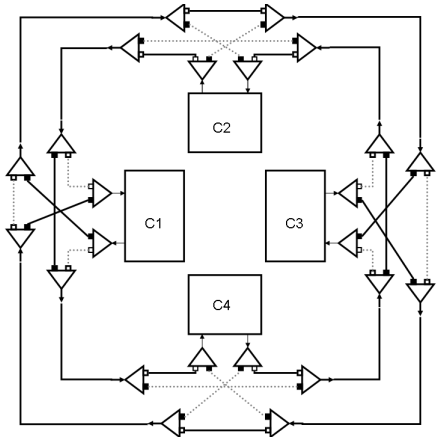

Figure 4: C1 doubly linked to C3, (and C2 to C4)

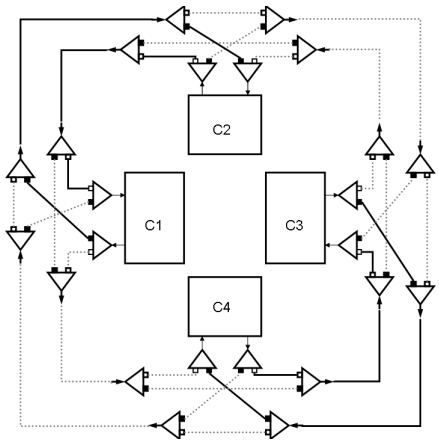

Figure 5: C1 doubly linked to C2, (and C3 to C4)

Definition 12 (Kahn-extended Event Graph) *A* Kahn-extended Event Graph (KEG) *is a structure* $E =< \mathcal{C}, \mathcal{M}, \mathcal{S}, \mathcal{P}, Sched, M >$*, together with several auxiliary functions listed below, such that:*

- $\mathcal{C}$ *is a finite set of* computation nodes*;*
- $\mathcal{S}$ *is a finite set of* select nodes*;*
- $\mathcal{M}$ *is a finite set of* merge nodes*;*
  We let $\mathcal{N} = \mathcal{C} \cup \mathcal{S} \cup \mathcal{M}$ *be the set of all nodes (or transitions).*
- $\mathcal{P} \subset \mathcal{N} \times \mathcal{N}$ *is a finite set of buffer channels (or places);*
- $InM_1, InM_0, OutM$ *are three functions:* $\mathcal{M} \longrightarrow \mathcal{P}$ *defining the three connections points of a merge node. We require:*
  $m = Source(p) \Leftrightarrow p = OutM(m)$,
  $m = Target(p) \Leftrightarrow (p = InM_0(m) \vee p = InM_1(m))$, *and* $InM_0(m) \neq InM_1(m)$*; this ensures that merge nodes are properly connected to channels according to their arity and functionality;*
- $InS, OutS_0, OutS_1$ *are three functions:* $\mathcal{S} \longrightarrow \mathcal{P}$ *defining the three connections points of a select node. We ask similar properties as in the merge case (details left to reader);*
- $SwitchCond$ *is a function from* $(\mathcal{M} \cup \mathcal{S}) \longrightarrow \mathbb{B}^{\mathbb{N}}$*;*
- $M \in (\mathcal{P} \longrightarrow \mathbb{N})$ *assigns an (initial) token marking to places.*

We note $n = Source(p)$ *and* $n' = Target(p)$ *when* $p = (n, n')$*. Conversely we note* $In(n) = \{p \in \mathcal{P} / \exists n',\ p = (n, n'\}$ *(and similarly for* $Out(n)$*).*

We shall assume that $\forall f \in \mathcal{M} \cup \mathcal{S},\ |SwitchCond(f)|_1 = |SwitchCond|_0 = \infty$*, so that switching conditions are fair.*

It should be noted that the only non-determinism allowed in the system comes from the internal switch condition patterns at merge and select nodes. As it is purely internal, behaviours shall remain monotoneous and continuous as previously. The relation with Kahn networks may remain rather obscure here. We go through the exercise of encoding local Kahn process component in our framework in annex at the end of this article.

Definition 13 (KEG firing rule) *The extension here should be rather obvious.*
The firing conditions for computation nodes stays unchanged (they consume one token in each of their incoming places, and produce one in each of their outgoing places);
Select node s *fires by consuming the first letter* x *of* $SwitchCond(s)$ *and a token in* $InS(s)$*, producing a token in place* $OutS_x(s)$
Merge node m *fires by consuming the first letter* x *of* $SwitchCond(m)$ *together with a token in* $InM(s)$ *(which must exists), and then produces a token in* $OutS_x(s)$

We extend the notion $E \xrightarrow{T} E'$ *to the case where* E *is a KEG and* T *is a subset of* $\mathcal{C} \cup \mathcal{S} \cup \mathcal{M}$*.*

Merge and select nodes are internal switches, subject to transformation. We now provide an equivalence definition for KEGs to behave the same on computation nodes.

Definition 14 *Let E, E' be two KEGs with same computation nodes. We say that E' em subsumes computationally E, noted $E \prec E'$, if for all firing sequence $E = E_0 \xrightarrow{T_1} E_1 \dots E_i \xrightarrow{T_{1+1}} E_{i+1}$ there is a matching firing sequence $E' = E'_0 \xrightarrow{T'_1} E'_1 \dots E'_i \xrightarrow{T'_{1+1}} E'_{i+1}$, such that $\forall i, T_i \cap \mathcal{C} = T'_i \cap \mathcal{C}$.*
We say that E and E' are computationally equivalent, noted $E \sim E'$ when $E' \prec E$ and $E \prec E'$

The relation $E \prec E'$ expresses the fact that E can perform the same computations as E', but at a slower rate. $E \sim E'$ can in fact be characterize by the fact that the two graphs have the same asap firings, when considering computations only.

Definition 15 (K-periodic Kahn-extended Event Graph) *We call a KEG* k-periodic *(KpKEG) iff all SwitchCond patterns are k-periodic words.*

The firing rule for KEGs is (as usual) composed from the firing rules of each operator, with a notion of asap firing associated.

One can also consider expressions $merge(c, w_0, w_1)$ and $select_i(c, w)$, $i = 0, 1$ as flow transformation operators. The merge operator enjoy (trivial) identities, and preserve k-periodicity:

Property 3

$$merge(\mathbb{1}, w_0, w_1) = w_1$$
$$merge(\bar{b}, w_0, w_1) = merge(b, w_1, w_0)$$

c, w_0, w_1 k-periodic $\Rightarrow$ $merge(c, w_0, w_1)$, $select_i(c, w)$ k-periodic.

The former definition of firings work in an asynchronous setting. In a synchronous setting we shall generally assume that merge and select node firings are instantaneous/combinatorial. The same expansion of latencies on computations and places can be performed so as to obtain a KEG where all nodes have instantaneous firings, and where only latencies of size 1 and 0 are applied to places/channels. Under the assumption that there is no 0-latency loop in the graph on can define the maximal asap firing (where all nodes are executed until stabilization). Because cycles are now somehow dynamic because of the changing switching patterns there may be finer criteria for sound asap execution without divergence or combinatorial loops (we are currently investigating the issue).

3.2 Abstraction of KEGs into SDF

We now define an abstraction function of a KpKEG into a SDF graph. It is obtained by forgetting the precise switch pattern over a period, to retain only the proportions under

which Selects and Merges consume or produce from and to which branch. KpKEG will then inherit the well-balanced conditions and safety properties from SDF theory. Liveness, in turn, will be even more complex to characterize, as it really depends also on the precise switching patterns.

Definition 16 (KpKEG abstraction to SDF)
Given $K =< \mathcal{C}, \mathcal{M}, \mathcal{S}, \mathcal{P}, Sched, M >$ *a KpKEG, we build its SDF abstraction* $< \mathcal{C}', \mathcal{P}', W, M >$ *as follows:*

- $\mathcal{C}' = \mathcal{C} \cup \mathcal{C}_{\mathcal{M}} \cup \mathcal{C}_{\mathcal{S}}$, *where* $\mathcal{C}_{\mathcal{M}}$ *and* $\mathcal{C}_{\mathcal{S}}$ *are two sets of "fresh" names in one to one relation with* $\mathcal{M}$ *and* $\mathcal{S}$ *respectively;*
- $\mathcal{P}' = \mathcal{P}$ *in which nodes in* $\mathcal{C}_{\mathcal{M}}$ *and* $\mathcal{C}_{\mathcal{S}}$ *have been substituted for nodes in* $\mathcal{M}$ *and* $\mathcal{S}$*;*
- W *the weight function is defined for each* $p \in \mathcal{P}'$ *as:*

  if $Source(p) \in \mathcal{C}$ *then* $W_1(p) = 1$
  if $Target(p) \in \mathcal{C}$ *then* $W_2(p) = 1$
  and if $Source(p)$ *was a merge node in* K*:*
  if $p = InM1(Source(p))$ *then* $W_1(p) = |Sched(p)|_1$
  if $p = InM0(Source(p))$ *then* $W_1(p) = |Sched| - |Sched(p)|_1$
  if $p = OutM(Source(p))$ *then* $W_1(p) = |Sched|$
  ... and similar definitions for when $Target(p)$ *was a merge, and either end of* p *was a select node.*

  The idea in the previous definitions is that, while plain computations just produce as many tokens as they consume on each channel, the production/consumption rate of a select or merge node can only be considered at the level of its switching pattern period.

Property 4 *A KpKEG* K *is* safe *iff its SDF abstraction is.*

The proof should be obvious by construction.

Property 5 *A KpKEG* K *is* live *when its SDF abstraction is.*

The proof here uses the fact that the SDF abstraction will allow each fireable merge and select node to progress by one full period of the switching pattern, independently of the rest of the nodes, as the token allocation required by the SDF to be live provides just enough tokens for this.

The former results may be seen as the main positive results of the paper. Still, the liveness condition is sufficient, but not always necessary. Indeed, sensible schedulings may allow the tokens to flow and progress in a more interleaved fashion, so that the system may be in fact remain live with less tokens. This greatly extend the issue we already mentioned of finding efficiently a minimal live marking already for SDF graphs.

In the sequel we shall investigate semantic-preserving transformations that may lead to optimizations of merge/select nodes towards more obvious and efficient scheduling pattern. This is still much ongoing work.

3.3 Composing switch patterns for `Select/Merge` permutations

Select and Merge operators shall be combined in the system description. Often the topology of this switching network could be modified by permuting or rearranging the configurations of merges and selects, possibly optimizing the buffering allocation needed for values represented as tokens. Analogies can be drawn with logical gate optimization and logic synthesis as used in hardware design. On the theoretical side one can also hope to reach canonical forms in some restricted cases, which then help greatly in proofs of properties.

The transformations will be based on appropriate matching transformations of the switching patterns. We introduce specific *On* and *When* operators which will be used to define them, and we study some of their algebraic properties. A previous version of *On* was introduced in [6], and the *When* operator is reminiscent of one under the same name used in synchronous reactive languages, but here adapted to our setting.

3.3.1 On and When operators

Definition 17 ("On" and "When" operators) *Let u, v be infinite binary words in $\mathbb{B}^{\mathbb{N}}$. The operators $u\,on\,v$ and $u\,when\,v$ are inductively defined as:*

$$\begin{aligned}(\mathbb{0}.u)\,on\,v &= \mathbb{0}.(u\,on\,v)\\ \forall\, x \in \mathbb{B},\ (\mathbb{1}.u)\,on\,(x.v) &= x.(u\,on\,v)\\ \forall\, x \in \mathbb{B},\ (x.u)\,when\,(\mathbb{0}.v) &= u\,when\,v\\ \forall\, x \in \mathbb{B},\ (x.u)\,when\,(\mathbb{1}.v) &= x.(u\,when\,v)\end{aligned}$$

The intuition behind these operators is as follows:
in $u\,on\,v$ the second argument v applies only on the active ($\mathbb{1}$) letters of the first argument u, and filters them according to its own selection pattern. It should be noted that u and $u\,on\,v$ "grow" synchronously in length, while v is consumed only at the pace of the $|u|_1$ only. Note that $u\,on\,v$ is a subflow of u;
conversely, $u\,when\,v$ preserves only those values in u at the instants selecting by the activity in v. Thus the result "grows" at the pace of $|v|_1$, while u and v are consumed synchronously. This operator is mostly meant to be applied under the conditions that u is a subflow of v; it then represents the erasing needed on the $\mathbb{1}$s in v to obtain u.

These operators enjoy a number of algebraic properties, some of which show the somehow mutually reverse effect of the two operators.

Property 6

$$\begin{aligned}
(u\,on\,v) &\sqsubseteq u \\
(u\,on\,\mathbb{1}^{\omega}) &= u \\
u\,when\,u &= 1^{|u|_1} \\
(|u|_1 = \infty) &\Rightarrow (u = (v\,on\,u)\,when\,v) \\
(u \sqsubseteq v) &\Rightarrow (u = v\,on\,(u\,when\,v)) \\
rate(u\,on\,v) &= rate(u).rate(v) \\
(u \sqsubseteq v) &\Rightarrow (rate(u\,when\,v) = rate(u)/rate(v))
\end{aligned}$$

Property 7 *u, v k-periodic $\Rightarrow$ $(u\,on\,v)$, $(u\,when\,v)$ k-periodic.*

3.4 Transformation identities on Merges and Selects

We now provide local transformations on *merge*s and *select*s, which respects the flow semantics. The switching condition streams are transformed using *On* and *When* constructs.

Property 8 *The transformations depicted in figures 6 and 7, with the corresponding relevant transformations on switching condition streams, preserve computation-equivalence.*

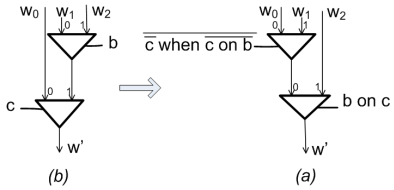

Figure 6: Permuting merges

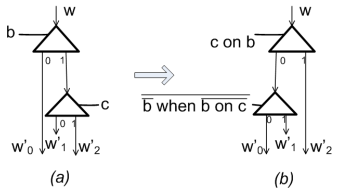

Figure 7: Permuting selects

The next transformation we shall consider introduces more tricky issues. It consists in the splitting of a common link, as displayed in figure 8. If a common link is shared (on the left side), then tokens may stack up with flows w_i, awaiting for the other side to take its turn. When the links are split more concurrency is allowed; but then the number of nodes requested may grow in complexity (each time with a cartesian product of former node couples).

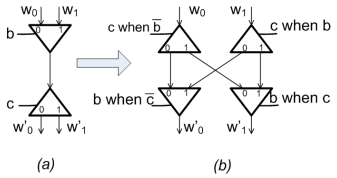

Figure 8: "De-sharing" an edge

The trick here is that there is a possibility that the configuration on the left directs the flow w_0 entirely towards w'_0, and reciprocally w_1 entirely towards w'_1. Then the right transformation provides two fully independent flows (following the vertical lines), and no activity rates need to be maintained between them. In other words, while the graph seems connected locally, all four switch patterns are unidirectional (uniformly $\mathbb{1}^\omega$), and the diagonal place links are never used, should be erased, unveiling the lack of connectivity. In the sequel we shall assume that nodes remain fair during the transformations or are simplified, and that the resulting graph remains strongly connected as a whole.

Property 9 *Let E be a strongly connected, live KEG, and let $E|prime$ be obtained from E by a transformation of the type displayed in figure 8. Then $E \prec E'$.*

What this says is that the transformation allows more orderings, and that liveness may be obtained sometimes with less tokens after transformation, but this is harmless. In fact the useful (and more problematic) transformation would consist in performing the converse transformation (from right to left), which amounts to sharing a unique channel link for performing data transport from several computation locations to others. This of course requires careful scheduling analysis, to route adequately and efficiently the various flows.

It should also be noted here that the scheme of figure 8 is obviously not reversible in full generality, as depicted in figure 9, which shows possible behaviours on flow prefixes of length 2. The unshared version allows on each case the second token of an input flow to precede the first one from the other when being output. This cannot be matched in the shared link version.

The former transformations can be applied iteratively inside the graph to get rid of complex DAGs (when not comprising *computation nodes*, but only merge/select ones). Stated formally,

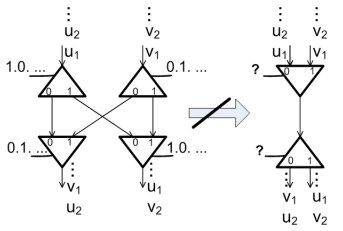

Figure 9: Sharing not aways feasible

Property 10 *Any KpKEG E can be transformed using merge/select permutations and de-sharing into a flow-equivalent one E' such that:*
$\forall s \in \mathcal{S}'$, $m \in \mathcal{M}'$ such that there is a path $\sigma_1 : OutS_0(s) \rightarrow InM_i(m)$ and a path $\sigma_2 : OutS_1(s) \rightarrow InM_j(m)$ with $i \neq j$, if the two paths do not *contain any computation node in $\mathcal{C}$, then $\exists p_1, p_2 \in \mathcal{P}$, $Source(p_1) = OutS_0(s)$, $Target(p_1) = InM_i(m)$, $Source(p_2) = OutS_1(s)$, and $Target(p_2) = InM_j(m)$.*

The proof works by induction on the lengths of the paths. What this proposition says is that complex connection DAGs (not involving synchronizing computation nodes),)with effective sharing of channel links, can be split to the point where only simple patterns of the shape in figure 10 remain.

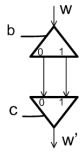

Figure 10: Simple (remaining) DAG

Again, several things are worth noticing on that simple pattern. First of course b and c must be of the same rate, or else tokens may accumulate in one of the inner channels. In the ideal case, the switch patterns of the upfront select and of the trailing merge will exactly match, so that the token exit the 2-node graph in the same order that they enter it (more precisely, one should also consider first purging potential initial tokens set in the place channels between the nodes). In this ideal case the graph can altogether be replaced by a single place channel. In less ideal cases of switching patterns, though, the tokens entering one channel may sometimes be bypassed by some selected faster along the other channel (as with local and express trains). For instance if $b = (0.1.1)^\omega$ and $c = (1.1.0)^\omega$, then every third token is kept in the "0-side" place channel to be bypassed by its two next followers

along the "1-side". Ultimately this may cause deadlock, since the token stuck in the inner channel may be the one that is expected to be "recycled" and re-enter next in the system.

One can use the information on (fixed, static) k-periodic switching patterns to establish minimal liveness conditions on initial token distributions in strongly connected KpKEG graphs. As in N-synchronous process theory one can also provide upper bounds to the content of places, again in the case of strongly connected graphs. The notion of order preservation may even be inserted, forcing the switches to open in the same direction so that tokens never get bypassed. In such a case (and disregarding different latencies for the time being) the pattern can be replaced by a simple (sequential channel), on which token routes are smoothly interleaved.

Next we provide a notion of order-preservation, which ensures that no token can bypass another one in a DAG rally, using alternative paths. We plan to use this notion to strenghten property 10 up to strong equivalence, and obtain canonical forms of graphs (in the order-preserving case of course).

3.5 Order preservation in token flows

As we have previously seen, a crucial aspect is that tokens following distinct routes between the same pair of computation nodes may bypass one another. We now provide means to characterize such phenomena (so that can can then be forbidden by excluding schedules that would display them). In the future we plan to extend our studies so as to encompass systems where bypasses are allowed, but in a bounded and predictable manner.

Definition 18 *We call a relation* $r \subseteq (\mathbb{N}\times)\mathbb{N}$loosely monotonic *whenever* $\forall (i_1, j_1), (i_2, j_2) \in r$, *not* $((i_1 < i_2)$ *and* $(j_2 < j_1))$.

In the sequel we shall restrict ourselves to KEGs without graph cycles involving only merge and select nodes (so each cycle must contain at least one computation node).

We shall first associate to every directed path between computation nodes $c$and c' a loosely monotonic relation $\curvearrowright$. A couple $(i \curvearrowright j)$ will represent how a token produced by the i^{th} firing of c will become the token consumed in the j^{th} firing of C'. The relation will be obtained by composition of elementary ones generated from merge, select and place elements. We shall split the sets $\mathcal{S}$ and $\mathcal{M}$ of select and merge nodes into $\mathcal{S}_0$, $\mathcal{S}_1$, $\mathcal{M}_0$, and $\mathcal{M}_1$ respectively, so that the node acknowledges which branch of the switch pattern is followed in the path.

Definition 19 (Token flow order relation) *We let* Σ *be the set of all simple paths in* E *(excluding paths containing a loop). A typical element in* Σ *is of the form* $\sigma = n_0.p_1.n_1. \ldots .n_l$, *with* $n_i \in \mathcal{C} \cup \mathcal{S}_0 \cup \mathcal{S}_1 \cup \mathcal{M}_0 \cup \mathcal{M}_1$ *and* $p_i \in \mathcal{P}$. *Single nodes are themselves paths of length 0. We shall associate to each path* σ *a monotone relation* $\curvearrowright_\sigma \subseteq (\mathbb{N} \times \mathbb{N})$ *obtained by composition of relations from the elementary relations of nodes and places (i.e.* $\curvearrowright_\sigma = \curvearrowright_{n_0} \circ \curvearrowright_{p_1} \circ \ldots \curvearrowright_{n_l}$, *where* $r \circ r' = \{(i,k) / (i,j) \in r\, (j,k) \in r'\}$*).*

The generator relations are themselves defined as follows:

- $\forall c \in \mathcal{C}$, $\forall i \in \mathbb{N}$, $i \curvearrowright_c i$;
- $\forall p \in \mathcal{P}$, $\forall i \in \mathbb{N}$, $i \curvearrowright_p (i + m_0(p))$, *where* $m_0(p)$ *is the initial marking of that place;*
- $\forall s \in \mathcal{S}_0$, $\forall i \in \mathbb{N}$, $[\overline{CondSwitch(s)}]_i \curvearrowright_{s_0} i$;
- $\forall s \in \mathcal{S}_1$, $\forall i \in \mathbb{N}$, $[CondSwitch(s)]_i \curvearrowright_{s_1} i$;
- $\forall m \in \mathcal{M}_0$, $\forall i \in \mathbb{N}$, $i \curvearrowright_{m_0} [\overline{CondSwitch(m)}]_i$;
- $\forall m \in \mathcal{M}_1$, $\forall i \in \mathbb{N}$, $i \curvearrowright_{m_1} [CondSwitch(m)]_i$;

In the previous definition the transformation induced by places is that tokens entering it get preceded by the ones originally residing there (in the initial marking). The transformations performed by the merge/select switches is that tokens gets numbererd by their rank in the switching pattern for that specific direction (remember that $[w]_i$ represents the index of the $i^{th}\mathbb{1}$ in the infinite binary word w).

Concerning the shapes of the relations, it could be noted that computation nodes contribute total bijective function, places and merges contribute total injective functions, while selects contribute partial functions that are onto (the converse relation is again a total function). More complex relations will arise when considering the several token flows running in parallel or alternative modes between nodes.

Definition 20 (Order preservation) *We call a KEG* E order-preserving *iff* $\forall c, c' \in \mathcal{C}$ *(possibly equal),* $\forall \sigma_1, \sigma_2 \in \Sigma$, $\sigma_1 \neq \sigma_2$, *then* $\curvearrowright_{\sigma_1} \cup \curvearrowright_{\sigma_2}$ *is itself a loosely monotonic relation.*

This definition says exactly that an indexed token can never bypass another from the same source. Order-preserving is a strong indication that schedules are indeed expecting tokens in the good order that they will arrive. We conjecture that order-preserving should make stronger the result of property 9.

4 Conclusion

Timed Event Graphs form a Model of Computation where Concurrency theory meets Scheduling theory, as activation patterns for computations can effectively be computed as k-periodic regimes on formal models. We dealt with the question of enlarging the framework with some amount of switching control, while trying to keep most of the benefits of the former approach. To this end we introduced merge and select nodes, again with k-periodic switching patterns. The liveness and safety problem can be tackle in the enlarged setting, by abstraction of Kahn-extended Event Graphs into (weighted) SDF graphs. But the liveness result imposes that KEG nodes can be scheduled on a coarse-grain scale, where each node can perform for a whole period independently of other nodes. The real challenge is rather to figure how perceptive scheduling can ensure liveness in systems where interleaving of behaviors is mandatory.

In KEG models datapaths can easily be set and reset, or checked for consistency. Under static scheduling one can evaluate where (and to which amount) tokens may or need to accumulate, and whether order is preserved between alternative routes between two nodes. This should allow us to compute buffering size needed, as in [6].

As an exercise we also described how to encode simple finite-state algorithms written *à la Kahn* into the KEG model, justifying its naming.

Further Topics The current results can be extended in many aspects. We should combine the k-periodicity in latency time of Event Graphs and in conditional switches of KEGS, to obtain a general activation condition pattern in time and space. The computation of minimal liveness conditions, and the effective dimensioning of buffers should be made more explicit, including their algorithmic aspects. Conditions under which the sharing/unsharing transformations can be applied, and to what precise semantic preservation, are also topics of future work. The complexity of initialization phases in particular should be better characterized. But, as it stands, we hope that the modeling framework of KEGs can be the basis of a theory for formal modeling of switching networks together with timed computation nodes, supporting scheduling results of decidable nature.

References

[1] Pascalin Amagbegnon, Loic Besnard, and Paul Le Guernic. Implementation of the data-flow synchronous language signal. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 163–173, New York, NY, USA, 1995. ACM Press.

[2] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity: an algebra for discrete event systems.* John Wiley & Sons, 1992.

[3] Julien Boucaron, Jean-Vivien Millo, and Robert De Simone. Latency-insensitive design and central repetitive scheduling. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings. Fourth ACM and IEEE International Conference on*, pages 175– 183, Piscataway, NJ, USA, 2006. IEEE Press.

[4] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, University of California, Berkeley, CA 94720, 1993.

[5] Jacques Carlier and Philippe Chrétienne. *Problème d'ordonnancement: modélisation, complexité, algorithmes.* Masson, Paris, 1988.

[6] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks: a relaxed model of synchrony for real-time systems. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT*

symposium on Principles of programming languages, pages 180–193, New York, NY, USA, 2006. ACM Press.

[7] F. Commoner, Anatol W.Holt, Shimon Even, and Amir Pnueli. Marked directed graph. *Journal of Computer and System Sciences*, 5:511–523, October 1971.

[8] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP CONGRESS 74*. North-Holland publishing company, 1974.

[9] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of IFIP CONGRESS 77*. North-Holland publishing company, 1977.

[10] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE transactions on computers*, C-36(1):24–35, 1987.

[11] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, CA 94720, December 1995.

[12] Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, MIT, Cambridge, Massachusetts, USA, September 1973.

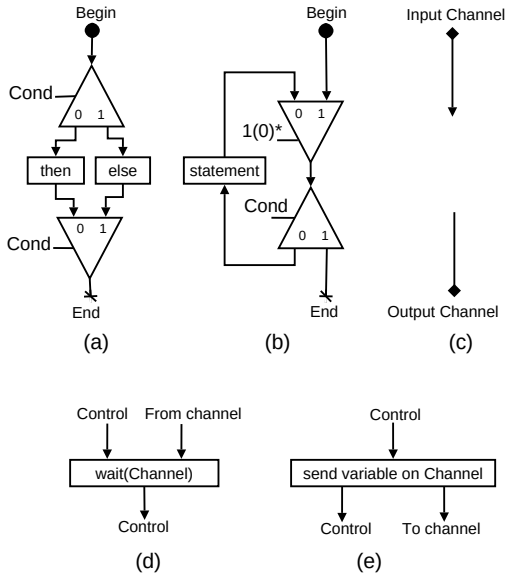

Figure 11: Translation of a) if-then-else composition b) repeat loop c) declaration of channel d) wait operation e) and send operation

A Appendix: Transforming Kahn process in KEG

A.1 Encoding of local deterministic components

The relation to Kahn networks remains rather obscure so far. In the original reference paper [8] Kahn process networks consist of local process nodes interacting through unbounded Fifo queues. Each node contains a sequential imperative programs, based on few syntactic constructs: local variable assignments and tests, sequential and if-then-else compositions, loops, with additional `wait` and `send` statements. `wait` represents a blocking read operation on a specified fixed channel, and `send` represents a non-blocking emission of a computed value, again in a fixed specified channel.

In our modelling we shall ignore and abstract away data values and local variables, and assume that the branching condition of each if-then-else test can be represented by a k-periodic pattern. This may seem very restrictive, but it should be remembered that such patterns are to be the result of computed scheduling decisions.

Figure 11 presents the graphic translation of elements of the imperative language used to described local process nodes. a) and b) represent structure of the language through the if-then-else composition and the repeat loop. c) show the declaration of input and output channel and finally (d and (e represent the wait and send operation. The *Control* input of

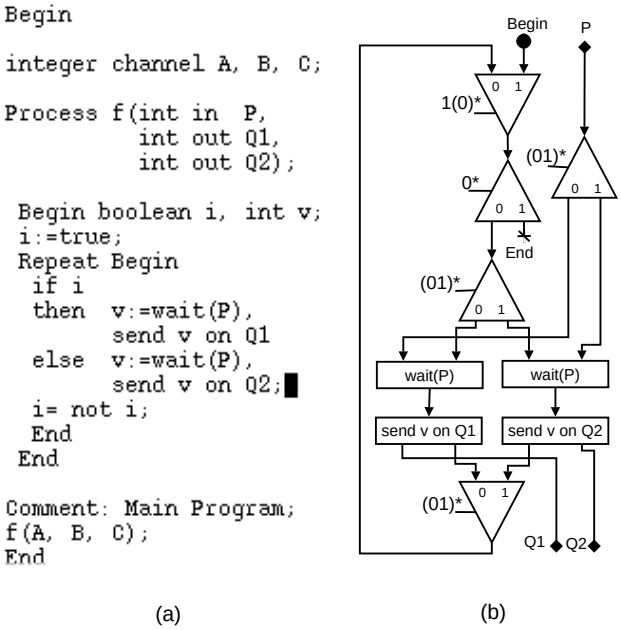

Figure 12: Translation of the textual style algorithm a) into KpKEG b)

both of them represent the sequentiality of operation. Wait operation needs in more a valid value/token from a channel to be computed.

The program in figure 12 a) is an simple example of process node. Figure 12 b) show its translation in a KpKEG. From the Begin label, the first merge and the first select are the structure of the repeat loops. the next select, the four computation nodes and the merge behind are the if-then-else structure. The last select on the up right, split the token flow arriving from the channel P and send them to the correct wait operation. If output channels like *Q1* or *Q2* have more than one producer, it should have some merge nodes to merge token flows.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399