Kahn-extended Event Graphs

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

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Kahn-extended Event Graphs

Julien Boucaron — Anthony Coadou — Benoît Ferrero — Jean-Viviven Millo — Robert de Simone

N° ????

May 2008

Thème COM

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

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

Kahn-extended Event Graphs *

Julien Boucaron , Anthony Coadou , Benoît Ferrero , Jean-Viviven Millo , Robert de Simone

Thème COM — Systèmes communicants
Équipe-Projet AOSTE

Rapport de recherche n° ???? — May 2008 — 52 pages

Abstract: Process Networks have long been used as formal Models of Computation in the design of dedicated hardware and software embedded systems and Systems-on-Chip. Choice-less models such as Marked/Event Graphs and their Synchronous Data Flow extensions have been considered to support periodic scheduling analysis. Those models do not hide dependency informations like regular sequential languages: they capture the communication topology through point-to-point channels. Those models are concurrent, formally defined, have a clear semantic but are limited due to static point-to-point channels. Then, further extensions such as Cyclo-Static Data Flow or Boolean-controlled Dataflow (BDF) graphs introduced routing switches, allowing internal choices while preserving conflict-freeness, in the tradition of Kahn Process Networks. We introduce a new model, which we term *Kahn-extended Event Graphs (KEG)*. It can be seen as a specialization of both Cyclo-Static and BDF processes. It consists merely in the addition of *Merge/Select* routing nodes to former Marked/Event Graphs; but, most importantly, these new nodes are governed by explicit (ultimately periodic) binary-word switching patterns for routing directions. We introduce identities on *Merge/Select* expressions, and show how they build a full axiomatization for the flow-equivalence between the computation nodes. The transformations carry a strong intuitive meaning, as they correspond to sharing/unsharing the interconnect links. Such interconnect defines each time a precise Network-on-Chip topology, and the switching patterns drive the traffic. One can also compute the buffering space actually required at the various fifo locations. The example of a Sobel edge filter is discussed to illustrate the importance of this model.

Key-words: Synchronous Data Flow, Kahn Process Network, Marked/Event Graph, Boolean-controlled Dataflow, Cyclo-Static Data Flow, System On Chip, Network On Chip, Liveness, Safety

* This work was partially supported by grants from ST Microelectronics and Texas Instruments inside the French CIM PACA regional collaborative project.

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Kahn-extended Event Graphs

Résumé : Les réseaux de processus (tels que les réseaux de Kahn) ont été utilisés comme modèles formels de calcul dans le cadre de la conception de logiciels et/ou matériels embarqués sur des systèmes sur puce. Les modèles dits "libre de choix" tels que les Marked/Event Graphs et l'extension Synchronous Data Flow peuvent être ordonnancés statiquement. Ces modèles ne cachent pas les dépendances tels que les langages séquentiels usuels: ils mettent en exergue la topologie de communication créée à l'aide de liaisons point à point. Ces modèles sont concurrents, formellement définis et disposent d'une sémantique claire grâce à ces précédentes liaisons point à point. D'autres extensions telles que le Cyclo-Static Data Flow ou le Boolean-controlled Dataflow (BDF) ont introduites des nœuds de routage permettant d'effectuer des choix internes tout en conversant la propriété d'absence de conflit, dans la tradition des réseaux de Kahn. Nous introduisons un nouveau modèle appelé *Kahn-extended Event Graphs (KEG)* qui peut être vu comme une spécialisation à la fois de Cyclo-Static et BDF. Nous introduisons des identités remarquables sur les expressions associées sur les *Merge/Select*, et montrons comment nous pouvons construire une axiomatisation pour l'équivalence de flot entre les nœuds de calcul. Ces transformations sont intuitives, elles correspondent à partager/éclater les liens de communication. Ces liens de communication définissent à chaque instant une topologie précise sur le réseau sur puce, et les conditions de routage dirigent le traffic. Nous pouvons aussi calculer les tailles des files nécessaires. L'exemple d'un simple filtre de détection de contour sur une image (Sobel) est détaillé afin d'illustrer l'importance de ce modèle.

Mots-clés : Synchronous Data Flow, Kahn Process Network, Marked/Event Graph, Boolean-controlled Dataflow, Cyclo-Static Data Flow, System On Chip, Network On Chip, Liveness, Safety

1 Introduction

Process networks (Petri Nets (PN) [20], Kahn Process Networks (KPN) [13]) provide models that are simple enough to conduct mathematical analysis, while supporting a clear operational semantic interpretation so to be used as abstract specifications. Examples range from Marked/Event Graphs (MEG) [9, 5] and their Synchronous Data Flow (SDF) [18, 17] extensions to more control-oriented graphs such as Cyclo-Static Data Flow (CSDF) [3] and Boolean-controlled Dataflow networks (BDF) [6].

The striking feature common to MEG and SDF models is that they offer only a fixed uniform data flow. In a graph where computation nodes are connected by directed point-to-point channels, all global executions essentially consist of the same partial order of individual computations, only possibly sooner or later. SDF graphs mainly add *weights* on channel ends, so that data are not consumed or produced individually, but as a fixed number all at one each time.

Those models have only static topology with only point to point communication channels. However more complex applications and limited ressources enable the need to have alternating routes for communications, in order to reuse components for instance. Moreover a lot of architectures such as FPGA, emerging NoCs on complex SoCs and recent processors such as the Cell [16] or the Tile64 [23] push further on the need of being able to describe an application with dynamic routes while being able to ensure safety, liveness, and equivalence of behavior with respect to a given specification.

Allowing multiple routes for data while preserving the global behavioral determinism whatever the communication speeds was previously studied as *monotonicity* [13] or *confluence* [19]. The usual restriction is to disallow external nondeterminism, in which the data availability (or not) on incoming channels can be tested as the cause of the local behavior. Instead, in KPN-style formalisms, the local component decides internally what the next action will be, including production and consumption on channels.

Among the attempts at defining such models, CSDF and BDF, both extensions of SDF, follow this discipline. In CSDF, each computation component goes cyclically through a number of states. Each state defines precisely a subset of input and output channels which shall be impacted at that stage: data will be consumed from indicated input channels, and produced to specified output channels as well. In BDF, there are two kinds of nodes introduced in addition to the previous computation nodes: a *Select* and a *Switch* node type (that we will further call *Select* and *Merge* in our terminology). They act respectively as a demultiplexer, dispatching data from a single stream into two distinct output channels, and as a multiplexer, joining two streams into one. In BDF models, the routing conditions are left unspecified.

We introduce Kahn-extended Event Graphs (KEGs) as a mix of CSDF and BDF, which inherit the good aspects of both (and hopefully not the bad ones). They borrow the general simpler form of BDF, but adopt the restrictions that routing condition patterns be ultimately periodic (as in CSDFs). This explicit form allows to compute the average production/computation rate of data from

and to channels over a full period. This information, which was available in CSDFs but *not* in BDF graphs, allows to abstract them further into SDF representation, thereby making it possible to solve their so-called "balanced equations". Balance equations are important because they check whether the production and consumption of data on each channel are indeed balanced, so that the network is globally free from both starvation and data overflows, and instead remains safe in general PN technically sense. We provide identities to be able to permute *Select/Merge* nodes and to transform the network, while preserving the correction of the design.

More concretely, our approach shall be seen as a contribution to define "low-level" formal models needed for *architectural/ high-level synthesis*. Recent academic or commercial tools, like AutoPilot/xPilot [10], Compaan [15] or PICO Express [14] to cite just a few, rely on such intermediate representations to perform scheduling and optimization steps. We also draw inspiration from the work on n-synchronous processes [8]. We plan in the future to combine more closely ultimately periodic routing and scheduling, to figure out algorithmic techniques and heuristics for combined optimization of both.

Outline The next section recalls the basic definition of formally defined models: Marked Event Graphs, Synchronous Data Flow graphs, Cyclo-Static Data Flow and Boolean-controlled Dataflow graphs. Then, we provide some of notations and auxiliary definitions on ultimately k-periodic binary words, which will play a big role in the theory of our KEG model, which we detail immediately afterwards. We comment on its expressivity, and its abstraction into SDF which preserves dataflow ratios on full periods. After, we discuss how we can encode Kahn Process Networks in the KEG model. Next, we define an order relation over token flows (order preservation), and the *on* and *when* operators applying of *Select/Merge* switching conditions. Then, we refine the KEG model in a synchronous way, introducing k-periodic schedules. After, we introduce transformations on KEG that are order preserving, using the previous operators. We introduce the *expansion* process which is unsharing common links, using previous operators. Then, we discuss in detail a Sobel edge filter starting from a description as a loopnest, we build a KEG, and apply the expansion. Finally, we conclude and extend on on-going work.

2 Background definitions and former models

In this section, we recall briefly what are MEG, SDF, CSDF, BDF and their main properties. We also introduce some notations and definitions on binary sequences.

2.1 Useful notations

Let $\mathcal{N}$ be a finite set of *computation nodes* and $\mathcal{P} \subset \mathcal{N} \times \mathcal{N}$ be a finite set of *places* (or *channels*).

For n a computation node, we note $\bullet n$ (resp. $n\bullet$) the *precondition* of n (resp. the *postcondition* of n), defined as:
- $\bullet n = \{p \in \mathcal{P} \,/\, \exists\, n' \in \mathcal{N},\, p = (n', n)\}$
- $n\bullet = \{p \in \mathcal{P} \,/\, \exists\, n' \in \mathcal{N},\, p = (n, n')\}$

Similarly for p a channel we note $\bullet p$ (resp. $p\bullet$) the *source* of p (resp. the *target* of p), defined as:
- $\bullet p = \{n \in \mathcal{N} \,/\, \exists\, n' \in \mathcal{N},\, p = (n, n')\}$
- $p\bullet = \{n \in \mathcal{N} \,/\, \exists\, n' \in \mathcal{N},\, p = (n', n)\}$

2.2 Marked/Event Graph

Marked Event Graph (MEG) [11, 9] is a sub-class of Petri Nets [20]. We briefly recall its definition.

Definition 1 (Marked/Event Graph). *A* Marked/ Event Graph *is a structure* $\langle \mathcal{N}, \mathcal{P}, M_0 \rangle$*, where:*

- M_0 *is a function:* $\mathcal{P} \to \mathbb{N}$ *assigning an initial content of (abstract) data to each channel.*
- $\forall\, p \in \mathcal{P},\ |\bullet p| = |p\bullet| = 1$ *(i.e. there is only one consumer and only one producer:* conflict-free*).*

A computation node n is called *enabled* (or fireable, or executable) whenever the current data assignment provides at least one data/token in each input channel of $\bullet n$. The effect of the firing of n is to build a new data assignment by removing one data from each input channel, and adding one as a result of the computation in each output channel of $n\bullet$.

We can define simultaneous firing of computations (when enabled as a whole), and a notion of (fastest) *asap* computation, where all computation nodes are fired as soon as enabled.

Due to the structural property of the set $\mathcal{P}$, any MEG is *confluent*: for any set of enabled computation nodes, executing any of these one do not remove any other computation node than themselves in the enabled set. This means that any firing rule is giving a partial order of execution of computation nodes compatible with the partial order of execution of computation nodes generated by the asap firing rule.

Concerning MEGs with strongly connected components, results of [9] state that a strongly connected MEG is safe and live iff there is at least one data initially in each loop of the graph (here safety means that the number of data in the system remains bounded, and live means that each node can be fired indefinitely).

An example of MEG is shown in the figure 1 with an example of execution: computation node b fires, computation node a fires.

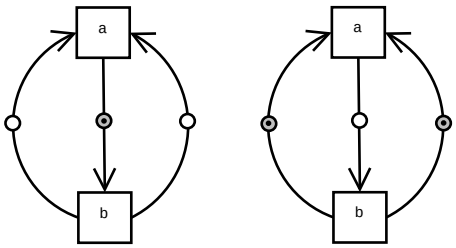

Figure 1: MEG Example

2.3 Synchronous Data Flow

Synchronous Data Flow (SDF) has been introduced by Edward A. Lee and David Messerschmitt [17], it is an extension of MEG associating at each input/output of a place a weight, which is the number of tokens consumed/produced respectively by the next/previous computation node.

Definition 2 (Synchronous Data Flow). *A* Synchronous Data Flow *[18] graph is a structure* $\langle \mathcal{N}, \mathcal{P}, M_0, \mathcal{W} \rangle$*, where:*

- $\langle \mathcal{N}, \mathcal{P}, M_0 \rangle$ *is a MEG.*
- $\mathcal{W}$ *is a weight function:* $\mathcal{P} \rightarrow \mathbb{N}^* \times \mathbb{N}^*$.

The weight defines an explicit number of tokens that is consumed (resp. produced) on the channel when the target (resp. source) node is fired. The definition of enableness and firing must be adapted accordingly to the weight function.

The questions of safety and liveness are somehow more complex for SDF graphs than simple MEGs. The master result in [17] provides a simple criterion for safety, solving so-called *balanced equations*: stating that the ratio of data flow streams must correspond in production and consumption. Liveness can then be checked by bounded-length firings.

The figure 2 shows an example of a SDF net.

Remark: a SDF where any computation node has the same weight on inputs and outputs is called an *Homogeneous SDF* and is equivalent to a MEG.

The next models will contain nodes which do *not* uniformly consume and produce on all their input/output channels. This introduces routing choices for data flow streams.

2.4 Cyclo-Static Data Flow

Cyclo-Static Data Flow graphs (CSDF) were introduced by Lauwereins and fellow co-authors [3]. They extend SDF graphs in the following way: the weights associated to the production and consumption of data from certain nodes may

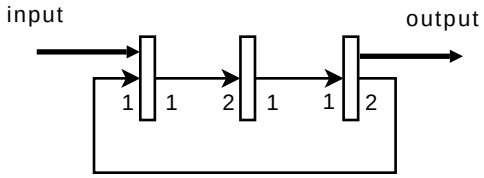

Figure 2: SDF Example

vary according to a cyclic pattern. Most results from SDF on balance equations carry to CSDF graphs.

Definition 3 (Cyclo-Static Data Flow). *A* Cyclo-Static Data Flow *is a structure* $\langle \mathcal{N}, \mathcal{P}, M_0, WeightList \rangle$ *where:*

- $\langle \mathcal{N}, \mathcal{P}, M_0 \rangle$ *is a MEG.*
- $WeightList$ *is function associating to each* $\bullet p$ *and* $p\bullet$*, with* $\forall\, p \in \mathcal{P}$*, a list of weights in* $\mathbb{N}$*.*

One of the main feature of CSDF is that the weight can be 0 whereas it is impossible in SDF for obvious reasons. Then, we can create *mux/demux*-like computation nodes using weight of value 0 on input/output of a computation node as shown on figure 3 (a).

CSDF is deterministic, and can be checked for safety (bounded buffering ressources) through an abstraction of the whole "period" of the Cyclo-Static net to a SDF net. If the obtained SDF net is safe then the Cyclo-Static net is.

The abstraction is constructed as follows:

- we recreate the same topology (computation nodes, places and arcs).
- for each computation node, for each input/output list of weights, we compute the SDF associated weight which is simply the overall sum of tokens consumed/produced multiplied by the length of the period (the smallest common multiple among lengths of lists).

The figure 3 (b) shows such abstraction as a SDF graph.

2.5 Boolean-controlled Dataflow

Boolean-controlled Dataflow (BDF) [6] graphs introduce two specific routing operators in addition to regular computation nodes. We call them *Merge* and *Select*, and they act as mux/demux to separate or reassemble data flow stream. They do this according to routing condition patterns which are left unspecified.

Definition 4 (Boolean-controlled Dataflow). *A* Boolean-controlled Dataflow *is a structure* $\langle \mathcal{N}, Se, Me, P, M_0, SwitchCond \rangle$ *where:*

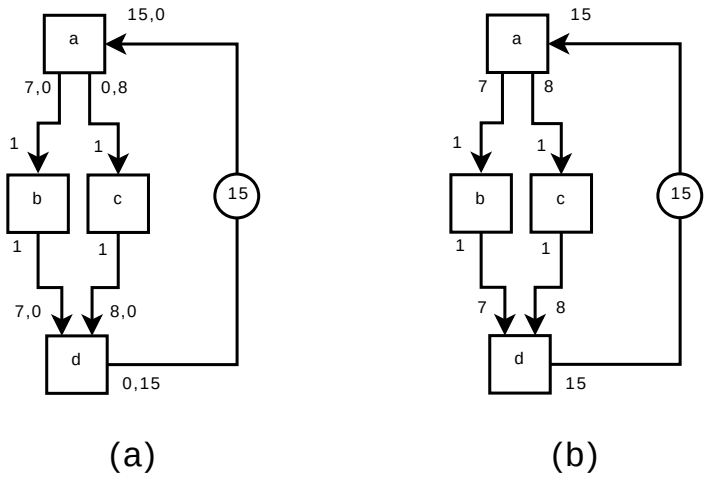

Figure 3: CSDF Example (a) and its SDF abstraction (b)

- *Se is a finite set of* Select *nodes such that* $\forall\, se \in Se : |se\bullet| = 2$ *and* $|\bullet se| = 1$.
- *Me is a finite set of* Merge *nodes such that* $\forall\, me \in Me : |me\bullet| = 1$ *and* $|\bullet me| = 2$.
- *P is a finite set of* places *such that* $P \subset Z \times Z$ *where* $Z = \mathcal{N} \cup Se \cup Me$ *and* $\forall\, p \in P : |p\bullet| = |\bullet p| = 1$.
- M_0 *is the initial marking of places* $M_0 : P \to \mathbb{N}$.
- *SwitchCond is a parametric function (can be data dependant) on* $Se \cup Me$.

Merge (resp. *Select*) nodes have exactly two distinct input (resp. output) channels, and exactly one output (resp. input) channel. The decision on which channel to consume from (resp. to produce onto) is taken internally, in some sort of *if-then-else* mechanism, since each channel is tagged as either the $1/then$ or the $0/else$ branch. But the predicate conditions are left unspecified, internal and can be data dependant. Because of this, BDF graphs loose the decidability property of dataflow stream balance equations, as one can not predict how often data will be sent or retrieved on which channels as shown in [6].

2.6 Periodic binary sequences

Now, we recall some definitions and notations on binary words coming from the n-synchronous theory [8].

Definition 5 (Binary words). *Let* $\mathbb{B} = \{0, 1\}$ *be our binary alphabet. We will further refer to this values as* characters, booleans *or* bits, *depending on the context.*

We call a binary word *any finite sequence of characters from* $\mathbb{B}$. *Let the Kleene closure* $\mathbb{B}^* = \bigcup_{n\in\mathbb{N}} \mathbb{B}^n$ *be the set of binary words. We note* ε *the empty word, such as* $\mathbb{B}^0 = \{\varepsilon\}$, *and let* $\mathbb{B}^+ = \mathbb{B}^* \setminus \{\varepsilon\}$ *be the set of non-empty binary words. We recall that* $\mathbb{B}^*$ *is a free monoid, with the concatenation as operation and* ε *as identity element.*

Note that the binary bitwise operators on binary words can only be applied on two words of same length. For convenience of writing, let w_i *be the* i^{th} *character of the binary word* w. *Let* $w_{head} = w_1$ *be its first character, and* w_{tail} *its other characters. That is to say:*

$$\forall\, w \in \mathbb{B}^+,\ \exists\, w_{head} \in \mathbb{B},\ \exists\, w_{tail} \in \mathbb{B}^*,\ w = w_{head}.w_{tail}$$

The length *of a binary word* w *is noted* $|w|$. *Moreover, let* $|w|_0$ *and* $|w|_1$ *be respectively the number of occurences of "0" and "1" in the binary word* w, *such that* $|w| = |w|_0 + |w|_1$.

We will further note $\mathbb{B}_p^k \subset \mathbb{B}^*$ *the set of words* w *with* $|w| = p$ *and* $|w|_1 = k$.

Definition 6 (Index). *Like [8], we note* $[w]_i$ *the position of the* i^{th} *occurrence of "1" in the binary word* w. *More formally:* $\forall\, w \in \mathbb{B}^*,\ \forall\, i \in [\![1, |w|]\!]$,

$$[w]_i = \begin{cases} \infty & \text{if } w = \varepsilon \\ 1 & \text{if } i = 1 \wedge w_{head} = 1 \\ [w_{tail}]_{i-1} + 1 & \text{if } i > 1 \wedge w_{head} = 1 \\ [w_{tail}]_i + 1 & \text{if } w_{head} = 0 \end{cases}$$

Example 1.

$$[1001101110]_1 = 1$$
$$[1001101110]_4 = 7$$

Definition 7 (Rate). $\forall\, w \in \mathbb{B}^*$,

$$\mathrm{rate}\,(w) = \begin{cases} 1 & \text{if } w = \varepsilon \\ \frac{|w|_1}{|w|} & \text{otherwise} \end{cases}$$

Example 2.

$$\mathrm{rate}\,(1001101110) = \frac{6}{10}$$

Definition 8 (Subflow). $\forall\, n \in \mathbb{N},\ \forall\, u, v \in \mathbb{B}^n$

$$u \sqsubseteq v \ \Leftrightarrow\ \forall\, i \in [\![1, n]\!],\ u_i \Rightarrow v_i$$

We can also express this property as:

$$u \sqsubseteq v \ \Leftrightarrow\ \exists\, w \in \mathbb{B}^n,\ u = v \wedge w$$

Thus, u *is said to be a* subflow *of* v.

Proposition 1. $\forall\, n \in \mathbb{N},\ \forall\, u, v \in \mathbb{B}^n$,

$$u \sqsubseteq v \ \Leftrightarrow\ \begin{cases} u \wedge v = u \\ u \vee v = v \\ \overline{v} \sqsubseteq \overline{u} \end{cases}$$

Proof. With the given definition, we know that each "1" of u matches a "1" of v. But a "0" in u may match as well a "0" or a "1" in v. Thus, with the conjunction operator the "0"s of u are paramount; the matching bits of v are paramount with the disjunction. The negation is a direct application. □

Proposition 2. $\forall\, n \in \mathbb{N},\ \forall\, u, v \in \mathbb{B}^n$

$$u \sqsubseteq v \wedge v \sqsubseteq u \ \Leftrightarrow\ u = v$$

Proof. It is obvious that:

$$\forall\, i \in [\![1, n]\!],\ (u_i \Rightarrow v_i) \wedge (v_i \Rightarrow u_i) \ \Leftrightarrow\ u_i = v_i$$

So $u = v$. □

Corollary 1. $\forall\, n \in \mathbb{N},\ \forall\ (u, v) \in \mathbb{B}^n \times \mathbb{B}^n$

$$u \sqsubseteq v \wedge |u|_1 = |v|_1 \ \Leftrightarrow\ u = v$$

Proof. If each "1" of u matches a "1" of v, and if there are as many "1" in u than in v, it is also obvious that it is the same with "0". Then, $u = v$. □

Definition 9 (Periodic binary sequences). *Let $\mathbb{B}^\omega$ be the set of* infinite binary sequences.

A periodic binary sequence *is the infinite repetition of a binary word, its periodic (or* steady*) part. We define the wider set $\mathbb{P}$ of* ultimately periodic sequences *(k-periodic sequences) [8]; they are periodic binary sequences possibly preceded by a fixed-length prefix such that:*

$$\mathbb{P} = \left\{ w = u.\,(v)^* \ \mid u \in \mathbb{B}^*,\, v \in \mathbb{B}^+ \right\}$$

We extend the rate notation with $rate\,(w) = rate\,(v)$. We note $\mathbb{P}_p^k$ the set of ultimately periodic sequences with $v \in \mathbb{B}_p^k$. We call k the periodicity *and p the* period *of such words.*

Because there is an infinity of representations of the same sequence, we decide to express any ultimately periodic binary sequence under its most factorized form, that is to say with the shortest prefix and period. Then, we can extend the notion of rate to ultimately periodic binary sequences, being equal to their $\frac{k}{p}$ ratio.

Example 3. $(01011)^* = 01011...101011010...$ *is a 3-periodic binary sequence with period 5, and so in $\mathbb{P}_5^3$. Its rate is equal to $\frac{3}{5}$.*

Note: we use here the term *sequence* instead of *word*, which may be commonly found in the literature with misuse of language. In compiler theory and formal languages, the length of a *word* (synonym of *sentence* or *string*) is *finite* per definition [1, 7, 12].

3 Kahn-extended Event Graph

In this section, we introduce Kahn-extended Event Graph (KEG) as a BDF-like extension of MEG with two additional nodes *Select* and *Merge* annotated with binary sequences. *Select* and *Merge* nodes are used to capture alternating routes as what can be achieved in the CSDF model. Then, we link this model to Kahn Process Network [13], due to the fact that the decision of taking a route on *Select/Merge* node is independent of data value and computed off-line. KEG is a special case of CSDF. Hence, a KEG is deterministic, and can be checked for safety (bounded resources) as in the CSDF case, we will show this in section 3.2.

3.1 Definition of KEG

Definition 10 (Kahn-extended Event Graph). *A* Kahn-extended Event Graph *is a structure* $\langle \mathcal{N}, Se, Me, P, M_0, SwitchCond \rangle$*, such that:*

- $\mathcal{N}, Se, Me, P, M_0$ *are borrowed from the BDF definition.*
- $SwitchCond$ *is a function* $Me \cup Se \rightarrow \mathbb{P}$*.*

We call a KEG k-periodic *due to the fact that all SwitchCond patterns are k-periodic sequences.*

Let us consider the intuitive example of figure 4.

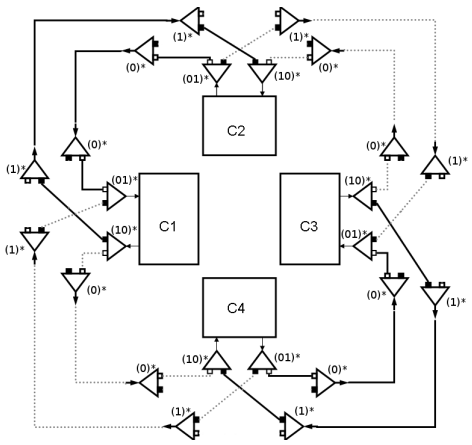

Figure 4: An example KEG: a simplified CELL network

It represents a (much simplified) multiprocessor interconnect topology in the way of the Cell chip [16]. 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 having two modes: $mode_1 = C_1 \leftrightarrow C_2$, $C_3 \leftrightarrow C_4$, and $mode_2 = C_1 \leftrightarrow C_4$, $C_2 \leftrightarrow C_3$. *Select* and *Merge* of outer rings have the same condition from one mode to the other one, since the length of each switching condition is one. On the other hand, if we look at *Select* and *Merge* just on the input/output of each C_x we have for each a switching condition of length two. When the *Select/Merge* fires, it takes the token on the associated input and send the token to the appropriate output, finally the switching condition is shifted on right. This KEG will alternatively execute $(mode_1.mode_2)^*$.

Definition 11 (KEG firing rule). *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 $x \in \mathbb{B}$ of $SwitchCond(s)$ and the input token, producing a token on the output x place.*

Merge *node m fires by consuming the first letter $x \in \mathbb{B}$ of $SwitchCond(m)$ together with the token on input x (which must exists), and then produces a token on the output.*

The former definition of firings works in an asynchronous setting. We will refine later the asynchronous firing rule as an asap firing rule, so that when a computation node (or *Select* or *Merge*) is enabled then it executes. Then, we can define a notion of throughput for the KEG, a notion of schedule (firing) for each computation node, *Select* and *Merge*, and we can compute the corresponding size for each fifo as will be shown in section 5.

This model do not have any explicit fixed latency attached on computation nodes or communication wires, but we can easily add them, in a similar way as found in [22]: for each node/wire having a latency greater than one, we segment it introducing additional nodes mimicking the original latency, while getting back to the model without any latency.

Merge and *Select* nodes are internal switches, subject to transformation.

In MEG/SDF, local computation nodes consume and produce systematically on all of their input and output channels respectively. Internal non-determinism is removed by requiring that channels have at most one input and output computation node. 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 KPN 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 MEG/SDF, 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).

The KEG combine both operational modes in a specific way; a KEG is determinist, behaviors remain monotonous and continuous as previously.

3.2 Abstraction of KEG into SDF

The problem of balanced token traffic so that consumption amounts to production on different routes, is again an issue for KEG. It is solved by abstraction reduction into SDF, safety preservation is already characterized by the balanced equations described in section 2.3. As in SDF, liveness is checked through bounded-length execution (in presence of safety).

We now provides the details of the abstraction itself.

Definition 12 (KEG abstraction to SDF). *Given a KEG we build its SDF abstraction* $\langle \mathcal{N}', \mathcal{P}, M_0, \mathcal{W} \rangle$ *as follows:*

- $\mathcal{N}' = \mathcal{N} \cup Se' \cup Me'$ *where* Se' *and* Me' *are new SDF nodes in one to one relation with* Se *and* Me*. We note* Se'_0 *and* Se'_1 *(resp.* Me'_0 *and* Me'_1*) the output (resp. input) sets associated to the* 0 *and* 1 *outputs (resp. inputs).*
- $\mathcal{P} = P$ *is the set of places.*
- M_0 *is the initial marking of places.*
- $\mathcal{W}$ *is the weight function associated to each place input and output:* $\forall\, p \in \mathcal{P}$*,*

  if $\bullet p \in \mathcal{N} : \mathcal{W}(p) = 1$.
  if $p\bullet \in \mathcal{N} : \mathcal{W}(p) = 1$.
  if $\forall\, x \in [0,1], \bullet p \in Se'_x : \mathcal{W}(p) = |SwitchCond|_x$.
  if $p\bullet \in Se' : \mathcal{W}(p) = |SwitchCond|$.
  if $\bullet p \in Me' : \mathcal{W}(p) = |SwitchCond|$.
  if $\forall\, x \in [0,1], p\bullet \in Me'_x : \mathcal{W}(p) = |SwitchCond|_x$.

The idea in the previous definitions is that, while plain $\mathcal{N}$ just produce/consume one token on each channel, the production/consumption rate of *Select/Merge* can only be considered at the level of its switching condition period.

An example of abstraction from a KEG graph to SDF is shown in figure 5 (a) and (b) respectively. Please note that for convenience of writing, we do *not* show the places of the KEG with void marking.

Property 1 (Soundness). *KEG abstraction to SDF is sound.*

Proof. In the previous definition there is a one to one mapping from each element in the original KEG graph to exactly one element in the SDF graph. Moreover, there is exactly the same one to one mapping for $\mathcal{P}$ and M_0 from the KEG graph

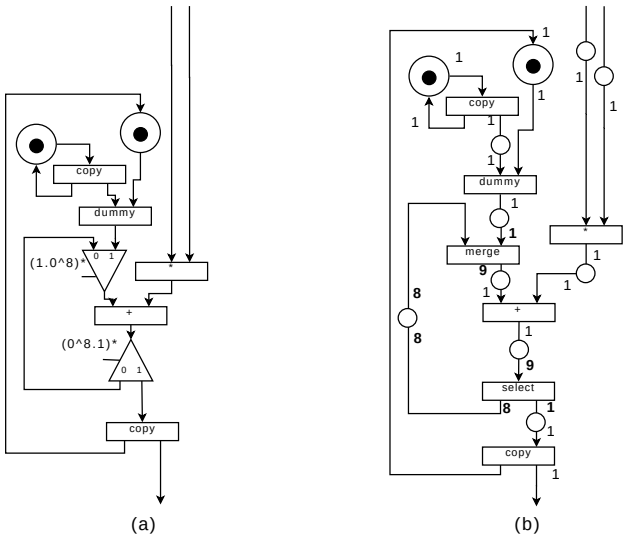

Figure 5: Abstraction from (a) a KEG to (b) SDF

to the SDF graph. Since both KEG/SDF are *confluent* we have the same set $\mathcal{P}$, and $\forall p \in \mathcal{P}$, $|\bullet p| = |p \bullet| = 1$, thus for each execution in the SDF graph there exists a corresponding set of executions in the KEG graph. The abstraction is sound. □

Corollary 2. *A KEG K is* safe *iff its SDF abstraction is.*

3.3 Encoding of KPN into KEG

In the original reference paper [13], Kahn Process Networks consist of local process nodes interacting through unbounded fifos. 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 6 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

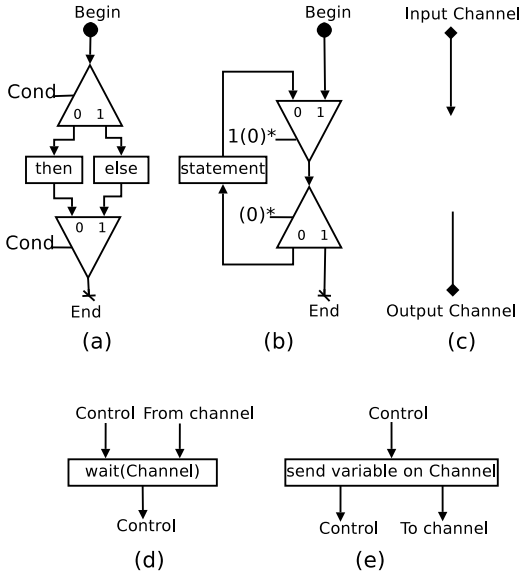

Figure 6: Translation of (a) if-then-else composition (b) repeat loop (c) declaration of channel (d) wait operation (e) and send operations

the wait and send operation. The *Control* input of both of them represent the sequentiality of operation. Wait operation needs an additional valid value/token from a channel to be computed.

The program in figure 7 (a) is a simple example of process node. Figure 7 (b) show its translation in a KEG. 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

4 On and when operators

This section introduce our formal description of switching conditions on *Select/Merge*. The following definitions are inspired by those given in [8], in order to formalize in the automata theory and boolean algebra the concepts of binary words and the operations upon them. We also demonstrate a few identities which will be useful later to simplify the computations and the network.

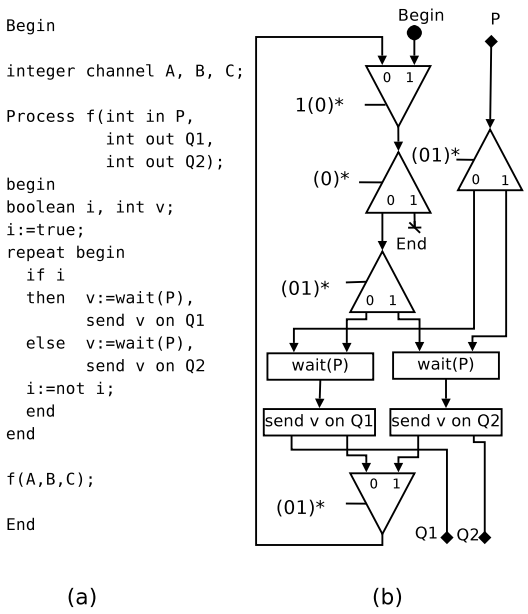

Figure 7: Translation of (a) the textual style algorithm into (b) KEG

4.1 Definitions

Definition 13 (*On* operator). *A previous version of* on *was introduced in [8]. It is recursively defined on binary words as:* $\forall\, n \in \mathbb{N},\ \forall\, u \in \mathbb{B}^n,\ \forall\, v \in \mathbb{B}^{|u|_1}$, *if* $|u|_1 = 0, \quad (u = 0^n)$

$$u \,\mathrm{on}\, v = u \,\mathrm{on}\, \varepsilon = u$$

otherwise,

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

Another way, maybe simpler, to express what does the on *operator is:* $\forall\, u \in \mathbb{B}^*,\ \forall\, v \in \mathbb{B}^{|u|_1},\ \exists\, w \in \mathbb{B}^{|u|}$,

$$u \,\mathrm{on}\, v = w = \begin{cases} w_{[u]_i} = v_i & \forall\, i \in [\![1, |v|]\!] \\ w_i = 0 & otherwise \end{cases}$$

Thus, $w \sqsubseteq u$.

Example 4.

$$001011 \,\mathrm{on}\, 101 = 001001$$

Definition 14 (*When* operator). *The* when *operator is roughly the same operator than the* filteredBy *operator defined in [2], despite our operator is primarily used in the case of branching conditions, not clocks. It is noted* ▼ *and recursively defined on binary words as:* $\forall\, n \in \mathbb{N},\ \forall\, u, v \in \mathbb{B}^n$,
if $n = 0$,

$$u\blacktriangledown v = \varepsilon\blacktriangledown\varepsilon = \varepsilon$$

otherwise,

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

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

$$u\blacktriangledown v = w = \left\{\forall\, i \in [\![1, |w|]\!],\, w_i = u_{[v]_i}\right\}$$

Example 5.

$$001001\blacktriangledown 001011 = 101$$

The intuition behind these operators is as follows:

- In $u \operatorname{on} v$, the second argument v applies only on the active ("1") bits of the first argument u, and filters them according to its own selection pattern. It should be noted that u and $u \operatorname{on} v$ "grow" together in length, while v is consumed only at the pace of the $|u|_1$ only. The assertion $|v| = |u|_1$ makes the definition consistent. Note that $u \operatorname{on} v$ is a subflow of u.
- Conversely, $u\blacktriangledown v$ preserves only those values in u at the instants selected 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 "1"s in v to obtain u.

The *on* and *when* operators are only defined over finite binary words. If we want to apply these operators over periodic binary sequences, we restrict the study to their periods. This way, we impose the fact that the sequences are well formed, that is to say there will not be an increasing asymptotic difference between the length of the words scanned or the number of "1"s matched, when the lengths go to infinity. Thus, the resulting sequence will be periodic, and equal to the infinite repetition of the result of the operation applied over the periods of the two operands.

4.2 On operator properties

Proposition 3 (Norms). $\forall\ (u, v) \in \mathbb{B}^n \times \mathbb{B}^{|u|_1}\ /\ n \geq |u|_1$,

$$\begin{aligned} |u \operatorname{on} v| &= |u| \\ |u \operatorname{on} v|_0 &= |u|_0 + |v|_0 \\ |u \operatorname{on} v|_1 &= |v|_1 \end{aligned}$$

Proof. The first equality is directly given by the definition: for each character of u, a character is put in the result, and $|u \operatorname{on} v| = |u|$.

The third equality comes from the fact that each "1"s of u is replaced by a character of v. Then, the result has as much "1" as v, and $|u \operatorname{on} v|_1 = |v|_1$.

The second equality is infered from the two others:

$$\begin{aligned} |u \operatorname{on} v|_0 &= |u \operatorname{on} v| - |u \operatorname{on} v|_1 \\ &= |u| - |v|_1 \\ &= |u|_0 + |u|_1 - |v|_1 \\ &= |u|_0 + |v| - |v|_1 \\ &= |u|_0 + |v|_0 + |v|_1 - |v|_1 \\ &= |u|_0 + |v|_0 \end{aligned}$$

□

Proposition 4 (Rate). $\forall\ (u, v) \in \mathbb{B}^n \times \mathbb{B}^{|u|_1}\ /\ n \geq |u|_1\,,$

$$\operatorname{rate}(u \operatorname{on} v) = \operatorname{rate}(u)\,.\,\operatorname{rate}(v)$$

Proof.

$$\operatorname{rate}(u \operatorname{on} v) = \frac{|u \operatorname{on} v|_1}{|u \operatorname{on} v|} = \frac{|v|_1}{|u|} = \frac{|v|}{|u|}.\frac{|v|_1}{|v|} = \frac{|u|_1}{|u|}.\frac{|v|_1}{|v|} = \operatorname{rate}(u)\,.\,\operatorname{rate}(v)$$

□

Proposition 5 (Subflow). $\forall\, u \in \mathbb{B}^*,\ \forall\, v \in \mathbb{B}^{|u|_1},$

$$u \operatorname{on} v \sqsubseteq u$$

Proof. By definition, each "1" of u is replaced by the corresponding bit of v, we have $\forall\, i \in [\![1, |u|]\!],\ [u \operatorname{on} v]_i \Rightarrow u_i$. Thus $u \operatorname{on} v \sqsubseteq u$. □

Corollary 3. $\forall\, u \in \mathbb{B}^*,\ \forall\, v \in \mathbb{B}^{|u|_1},\ \forall\, i \in \mathbb{N}^*,$

$$[u]_{[v]_i} = [u \operatorname{on} v]_i$$

Proof. By direct application of definitions 8 and 13:

$$u \operatorname{on} v \sqsubseteq u \quad \Leftrightarrow \quad \forall\, i \in \mathbb{N},\ \exists\, j \in \mathbb{N}\ /\ [u]_j = [u \operatorname{on} v]_i$$

where $j = [v]_i$. □

Proposition 6 (Idempotent elements).

$$\forall\, n \in \mathbb{N},\ 1^n \operatorname{on} 1^n = 1^n$$

Proof. By direct application of the definition of the *on* operator, the length of the right operand is equal to the number of "1"s in the left operand. Therefore, the only words with this property are those made only of "1"s. □

Proposition 7 (Identity elements). *Left identity:*

$$1^{|u|} \operatorname{on} u = u$$

Right identity:

$$u \operatorname{on} 1^{|u|_1} = u$$

Proof. On the one hand, it is obvious that, per definition, if each "1" of $1^{|u|}$ is replaced by the corresponding bit of u, the result will be equal to u. On the other hand, if each "1" of u is replaced by a "1" from $1^{|u|_1}$, the result will also be equal to u. □

Proposition 8 (Associativity). $\forall\, u \in \mathbb{B}^*,\ \forall\, v \in \mathbb{B}^{|u|_1},\ \forall\, w \in \mathbb{B}^{|v|_1},$

$$(u \operatorname{on} v) \operatorname{on} w = u \operatorname{on} (v \operatorname{on} w)$$

Proof. Given in [8]. □

Proposition 9 (Left-distributivity over bitwise operators). $\forall\, u \in \mathbb{B}^*,\ \forall\, v, w \in \mathbb{B}^{|u|_1},$

$$\begin{aligned} u \operatorname{on} (v \wedge w) &= (u \operatorname{on} v) \wedge (u \operatorname{on} w) \\ u \operatorname{on} (v \vee w) &= (u \operatorname{on} v) \vee (u \operatorname{on} w) \\ u \operatorname{on} (v \oplus w) &= (u \operatorname{on} v) \oplus (u \operatorname{on} w) \end{aligned}$$

Proof. By direct application of the definition, let us show per induction that the property on bitwise conjunction is true with $n = 0$:

$$u \operatorname{on} (\varepsilon \wedge \varepsilon) = u \operatorname{on} \varepsilon = u = u \wedge u = u \operatorname{on} \varepsilon \wedge u \operatorname{on} \varepsilon$$

Now, we suppose the property true at rank $n-1$ (the tails of the binary words), and we show its correctness for n. If $u_{head} = 0$,

$$\begin{aligned} u \operatorname{on} (v \wedge w) &= 0.\,(u_{tail} \operatorname{on} (v_{tail} \wedge w_{tail})) \\ &= 0.\,(u_{tail} \operatorname{on} v_{tail}) \wedge 0.\,(u_{tail} \operatorname{on} w_{tail}) \\ &= u \operatorname{on} v \wedge u \operatorname{on} w \end{aligned}$$

If $u_{head} = 1$,

$$\begin{aligned} u \operatorname{on} (v \wedge w) &= (v_{head} \wedge w_{head})\,.\,(u_{tail} \operatorname{on} (v_{tail} \wedge w_{tail})) \\ &= v_{head}.\,(u_{tail} \operatorname{on} v_{tail}) \wedge w_{head}.\,(u_{tail} \operatorname{on} w_{tail}) \\ &= u \operatorname{on} v \wedge u \operatorname{on} w \end{aligned}$$

Thus, we can conclude that the property is true for all n. The same inductive reasoning can be used to prove the correctness of the property on bitwise disjunctions. □

Proposition 10 (Negation). $\forall\, u \in \mathbb{B}^*,\ \forall\, v \in \mathbb{B}^{|u|_1}$,

$$\overline{u \operatorname{on} v} = \overline{u} \oplus (u \operatorname{on} \overline{v})$$
$$\overline{u \operatorname{on} v} = \overline{u} \vee (u \operatorname{on} \overline{v})$$

Proof.

$$\begin{array}{lrll} & u & = u \operatorname{on} 1^{|u|_1} & \text{by prop.7} \\ \Leftrightarrow & u & = u \operatorname{on} (v \oplus \overline{v}) & \\ \Leftrightarrow & u & = (u \operatorname{on} v) \oplus (u \operatorname{on} \overline{v}) & \text{by prop.9} \\ \Leftrightarrow & u \operatorname{on} v & = u \oplus (u \operatorname{on} \overline{v}) & \\ \Leftrightarrow & \overline{u \operatorname{on} v} & = \left(\overline{u} \wedge \overline{u \operatorname{on} \overline{v}}\right) \oplus (u \wedge (u \operatorname{on} \overline{v})) & \\ \Leftrightarrow & \overline{u \operatorname{on} v} & = \overline{u} \oplus (u \operatorname{on} \overline{v}) & \text{by prop.1 and 5} \end{array}$$

The second equality could be proved exactly in the same way. □

Corollary 4. $\forall\, u \in \mathbb{B}^*,\ \forall\, v \in \mathbb{B}^{|u|_1}$,

$$u \operatorname{on} \overline{v} = \overline{u \operatorname{on} v} \wedge u$$

Proof. Let $w = \overline{v}$. By the previous property (10), we have:

$$\begin{array}{lc} & \overline{u \operatorname{on} w} = \overline{u} \vee (u \operatorname{on} \overline{w}) \\ \Leftrightarrow & \overline{u \operatorname{on} \overline{v}} = \overline{u} \vee (u \operatorname{on} v) \\ \Leftrightarrow & \overline{\overline{u \operatorname{on} \overline{v}}} = \overline{\overline{u} \vee (u \operatorname{on} v)} \\ \Leftrightarrow & u \operatorname{on} \overline{v} = u \wedge \overline{u \operatorname{on} v} \end{array}$$

□

4.3 When operator properties

Proposition 11 (Right identity elements). $\forall\, u \in \mathbb{B}^*$,

$$u \blacktriangledown 1^{|u|} = u$$

Proof. By direct application of the definition,

$$u \blacktriangledown 1^{|u|} = v \ \Leftrightarrow\ \begin{cases} |v| = \left|1^{|u|}\right|_1 = |u| \\ \forall\, i \in [\![1, |u|]\!],\ w_i = u_i \end{cases}$$

because $\left[1^{|u|}\right]_i = i$. Thus $w = u$. □

Proposition 12 (Idempotent elements). $\forall\, n \in \mathbb{N}$,

$$1^n \blacktriangledown 1^n = 1^n$$

Proof. Per definition, the length of the result is equal to the number of "1"s in the right operand. The only binary words verifying this property are those made only of "1"s. □

Proposition 13. $\forall\, n \in \mathbb{N},\ \forall\, u, v \in \mathbb{B}^n,$

$$u \sqsubseteq v \ \Rightarrow\ v\blacktriangledown u = 1^{|u|_1}$$

Proof. By direct application of the definition of the *when* operator and proposition 1, v is sampled by the "1" in u. Each "1" in u matching a "1" in v, it is obvious that the result will be a sequence of "1"s. □

Proposition 14 (Rate). $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v \in \mathbb{B}^n\ /\ |v|_1 > 0,$

$$u \sqsubseteq v \ \Rightarrow\ \mathrm{rate}\,(u\blacktriangledown v) = \frac{\mathrm{rate}\,(u)}{\mathrm{rate}\,(v)}$$

Proof.

$$\begin{aligned} u \sqsubseteq v \ \Rightarrow\ u = v\,\mathrm{on}\,(u\blacktriangledown v) \quad &\Leftrightarrow \quad u \sqsubseteq v \ \Rightarrow\ \mathrm{rate}\,(u) = \mathrm{rate}\,(v\,\mathrm{on}\,(u\blacktriangledown v)) \\ &\Leftrightarrow \quad u \sqsubseteq v \ \Rightarrow\ \mathrm{rate}\,(u) = \mathrm{rate}\,(v)\,.\,\mathrm{rate}\,(u\blacktriangledown v) \\ &\Leftrightarrow \quad u \sqsubseteq v \ \Rightarrow\ \mathrm{rate}\,(u\blacktriangledown v) = \frac{\mathrm{rate}\,(v)}{\mathrm{rate}\,(u)} \end{aligned}$$

□

Proposition 15 (Right-distributivity over bitwise operators). $\forall\, n \in \mathbb{N},\ \forall\, u, v, w \in \mathbb{B}^n,$

$$\begin{aligned} (u \wedge v)\blacktriangledown w &= (u\blacktriangledown w) \wedge (v\blacktriangledown w) \\ (u \vee v)\blacktriangledown w &= (u\blacktriangledown w) \vee (v\blacktriangledown w) \\ (u \oplus v)\blacktriangledown w &= (u\blacktriangledown w) \oplus (v\blacktriangledown w) \end{aligned}$$

Proof. We show per induction that the property on bitwise conjunction is true. First, let suppose that $w = \varepsilon$. By the hypothesis, we also have $u = v = \varepsilon$. Then:

$$\begin{aligned} (\varepsilon \wedge \varepsilon)\blacktriangledown \varepsilon &= \varepsilon\blacktriangledown\varepsilon \\ &= \varepsilon \\ &= (\varepsilon\blacktriangledown\varepsilon) \wedge (\varepsilon\blacktriangledown\varepsilon) \end{aligned}$$

Now, we suppose the property true at rank $n-1$ (the tails of the binary words), and we show its correctness for n. If $w_{head} = 0$,

$$\begin{aligned} (u \wedge v)\blacktriangledown w &= (u_{tail} \wedge v_{tail})\blacktriangledown w_{tail} \\ &= (u_{tail}\blacktriangledown w_{tail}) \wedge (v_{tail}\blacktriangledown w_{tail}) \\ &= (u\blacktriangledown w) \wedge (v\blacktriangledown w) \end{aligned}$$

Else if $w_{head} = 1$,

$$\begin{aligned}(u \wedge v)\blacktriangledown w &= (u_{head} \wedge v_{head}) \,.\, ((u_{tail} \wedge v_{tail})\blacktriangledown w_{tail}) \\ &= (u_{head} \,.\, (u_{tail}\blacktriangledown w_{tail})) \wedge (v_{head} \,.\, (v_{tail}\blacktriangledown w_{tail})) \\ &= (u\blacktriangledown w) \wedge (v\blacktriangledown w)\end{aligned}$$

Thus the property is true. The same inductive reasoning can be used to prove the correctness of the property on bitwise disjunction. □

Proposition 16 (Negation). $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v \in \mathbb{B}^n$,

$$\overline{u\blacktriangledown v} = \overline{u}\blacktriangledown v$$

Proof.

$$\begin{aligned}& 1^{|v|_1} &&= 1^{|v|}\blacktriangledown v \\ \Leftrightarrow\ & 1^{|v|_1} &&= (u \oplus \overline{u})\blacktriangledown v \\ \Leftrightarrow\ & 1^{|v|_1} &&= (u\blacktriangledown v) \oplus (\overline{u}\blacktriangledown v) \qquad \text{by prop.15} \\ \Leftrightarrow\ & u\blacktriangledown v &&= 1^{|v|_1} \oplus \overline{u}\blacktriangledown v \\ \Leftrightarrow\ & \overline{u\blacktriangledown v} &&= \left(1^{|v|_1} \wedge (\overline{u}\blacktriangledown v)\right) \vee \left(0^{|v|_1} \wedge \overline{\overline{u}\blacktriangledown v}\right) \\ \Leftrightarrow\ & \overline{u\blacktriangledown v} &&= \overline{u}\blacktriangledown v\end{aligned}$$

□

Proposition 17. $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v \in \mathbb{B}^n$,

$$u \wedge v = 0^{|u|} \ \Rightarrow\ u\blacktriangledown (u \vee v) = \overline{v}\blacktriangledown (u \vee v)$$

Proof. Let us show by induction the correctness of the property. It is obvious that it is true at the starting point, where $u = v = \varepsilon$.

$$\varepsilon \wedge \varepsilon = \varepsilon \ \Rightarrow\ \varepsilon\blacktriangledown (\varepsilon \vee \varepsilon) = \overline{\varepsilon}\blacktriangledown (\varepsilon \vee \varepsilon)$$

Now, let suppose the property true for the tails of u and v, and let us show the correctness with the whole words.

$$\begin{aligned}& u \wedge v = 0^{|u|} \ \Rightarrow\ u\blacktriangledown (u \vee v) = \overline{v}\blacktriangledown (u \vee v) \\ \Leftrightarrow\ & ((u_{head} \wedge v_{head} = 0) \wedge \left(u_{tail} \wedge v_{tail} = 0^{|u_{tail}|}\right) \\ \Rightarrow\ & (u_{head}\blacktriangledown (u_{head} \vee v_{head})) \,.\, (u_{tail}\blacktriangledown (u_{tail} \vee v_{tail})) \\ =\ & (\overline{v_{head}}\blacktriangledown (u_{head} \vee v_{head})) \,.\, (\overline{v_{tail}}\blacktriangledown (u_{tail} \vee v_{tail})))\end{aligned}$$

Because we have supposed the correctness of:

$$u_{tail} \wedge v_{tail} = 0^{|u_{tail}|} \ \Rightarrow\ u_{tail}\blacktriangledown (u_{tail} \vee v_{tail}) = \overline{v_{tail}}\blacktriangledown (u_{tail} \vee v_{tail})$$

The problem is equivalent to:

$$u_{head} \wedge v_{head} = 0 \ \Rightarrow\ u_{head}\blacktriangledown (u_{head} \vee v_{head}) = \overline{v_{head}}\blacktriangledown (u_{head} \vee v_{head})$$

If $u_{head} \oplus v_{head} = 1$, then:

$$\begin{aligned} u_{head}\blacktriangledown(u_{head} \vee v_{head}) &= u_{head}\blacktriangledown 1 \\ &= u_{head} \\ &= \overline{v_{head}} \\ &= \overline{v_{head}}\blacktriangledown 1 \\ &= \overline{v_{head}}\blacktriangledown(u_{head} \vee v_{head}) \end{aligned}$$

Else if $u_{head} \vee v_{head} = 0$, then:

$$\begin{aligned} u_{head}\blacktriangledown(u_{head} \vee v_{head}) &= u_{head}\blacktriangledown 0 \\ &= \varepsilon \\ &= \overline{v_{head}}\blacktriangledown 0 \\ &= \overline{v_{head}}\blacktriangledown(u_{head} \vee v_{head}) \end{aligned}$$

Thus, the property is true for all u and v, which conclude the proof. □

Proposition 18. $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v, w \in \mathbb{B}^n$,

$$v \sqsubseteq w \quad \Rightarrow \quad u\blacktriangledown v = (u \wedge w)\blacktriangledown v$$

Proof.

$$\begin{aligned} & v \sqsubseteq w \ \Rightarrow\ (u \wedge w)\blacktriangledown v = (u\blacktriangledown v) \wedge (w\blacktriangledown v) \quad \text{by prop.15} \\ \Leftrightarrow\ & v \sqsubseteq w \ \Rightarrow\ (u \wedge w)\blacktriangledown v = (u\blacktriangledown v) \wedge 1^{|w|_1} \\ \Leftrightarrow\ & v \sqsubseteq w \ \Rightarrow\ (u \wedge w)\blacktriangledown v = u\blacktriangledown v \end{aligned}$$

□

4.4 Common properties

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

Proposition 19. $\forall\, v \in \mathbb{B}^*,\ \forall\, u \in \mathbb{B}^{|v|_1}$,

$$u = (v \operatorname{on} u)\blacktriangledown v$$

Proof. This property is a direct induction of the second wordings of both *on* and *when* operators. Let w be a binary word such that $w = v \operatorname{on} u$. Thus:

$$w = \begin{cases} w_{[v]_i} = u_i & \forall\, i \in [\![1, |u|]\!] \\ w_i = 0 & otherwise \end{cases}$$

which is another way to say that $u = w\blacktriangledown v$. So $u = (v \operatorname{on} u)\blacktriangledown v$. □

Proposition 20. $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v \in \mathbb{B}^n,$

$$u \sqsubseteq v \ \Leftrightarrow\ u = v \operatorname{on} (u \blacktriangledown v)$$

Proof. The proof is like the previous one, with a little difference: it is true *iff* u is a subflow of v ; that is to say, when none of the "1"s of u is discarded during the sampling by v. Otherwise, the equality $w_{[v]_i} = u_i$ would become false. □

Proposition 21 (Subflows). $\forall\, w \in \mathbb{B}^*,\ \forall\, u, v \in \mathbb{B}^{|w|_1},$

$$u \sqsubseteq v \ \Leftrightarrow\ w \operatorname{on} u \sqsubseteq w \operatorname{on} v$$
$$u \sqsubseteq v \ \Rightarrow\ u \blacktriangledown w \sqsubseteq v \blacktriangledown w$$

Proof. We first show the correctness of the property with the *on* operator.

$$\begin{aligned} u \sqsubseteq v \ &\Leftrightarrow\ (w \operatorname{on} u) \wedge (w \operatorname{on} v) = w \operatorname{on} (u \wedge v) \quad \text{by prop.9} \\ &\Leftrightarrow\ (w \operatorname{on} u) \wedge (w \operatorname{on} v) = w \operatorname{on} u \quad \text{by prop.1} \end{aligned}$$

$$\begin{aligned} u \sqsubseteq v \ &\Leftrightarrow\ (w \operatorname{on} u) \vee (w \operatorname{on} v) = w \operatorname{on} (u \vee v) \quad \text{by prop.9} \\ &\Leftrightarrow\ (w \operatorname{on} u) \vee (w \operatorname{on} v) = w \operatorname{on} v \quad \text{by prop.1} \end{aligned}$$

Then, one more time with prop.1, we have:

$$u \sqsubseteq v \ \Leftrightarrow\ (w \operatorname{on} u) \sqsubseteq (w \operatorname{on} v)$$

In a similar way, we show the correctness of the property with the *when* operator.

$$\begin{aligned} u \sqsubseteq v \ &\Leftrightarrow\ (u \blacktriangledown w) \wedge (v \blacktriangledown w) = (u \wedge v) \blacktriangledown w \quad \text{by prop.15} \\ &\Leftrightarrow\ (u \blacktriangledown w) \wedge (v \blacktriangledown w) = u \blacktriangledown w \quad \text{by prop.1} \end{aligned}$$

$$\begin{aligned} u \sqsubseteq v \ &\Leftrightarrow\ (u \blacktriangledown w) \vee (v \blacktriangledown w) = (u \vee v) \blacktriangledown w \quad \text{by prop.15} \\ &\Leftrightarrow\ (u \blacktriangledown w) \vee (v \blacktriangledown w) = v \blacktriangledown w \quad \text{by prop.1} \end{aligned}$$

Finally, with prop.1, we have:

$$u \sqsubseteq v \ \Leftrightarrow\ (u \blacktriangledown w) \sqsubseteq (v \blacktriangledown w)$$

□

Proposition 22. $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v \in \mathbb{B}^n,\ \forall\, w \in \mathbb{B}^{|u \wedge v|_1},$

$$(u \blacktriangledown v) \operatorname{on} w = ((u \wedge v) \operatorname{on} w) \blacktriangledown v$$

Proof.

$$\begin{aligned} v \operatorname{on} (u \blacktriangledown v) \operatorname{on} w \ &= \ (u \wedge v) \operatorname{on} w \quad \text{by prop.20} \\ \Rightarrow\ (v \operatorname{on} (u \blacktriangledown v) \operatorname{on} w) \blacktriangledown v \ &= \ ((u \wedge v) \operatorname{on} w) \blacktriangledown v \quad \text{by prop.21} \\ \Leftrightarrow\ (u \blacktriangledown v) \operatorname{on} w \ &= \ ((u \wedge v) \operatorname{on} w) \blacktriangledown v \quad \text{by prop.19} \end{aligned}$$

□

Proposition 23. $\forall\, v, w \in \mathbb{B}^*,\ \forall\, u \in \mathbb{B}^{|w|_1}$,

$$u\blacktriangledown\,(v\blacktriangledown w) = (w \,\mathrm{on}\, u)\blacktriangledown\,(v \wedge w)$$

Proof.

$$\begin{aligned}
u\blacktriangledown\,(v\blacktriangledown w) &= (u\blacktriangledown\,(v\blacktriangledown w)) \wedge 1^{|w|_1} \\
&= (u\blacktriangledown\,((v \wedge w)\blacktriangledown w)) \wedge ((v\blacktriangledown w)\blacktriangledown\,(v\blacktriangledown w)) \quad \text{by prop.13} \\
&= (u \wedge (v\blacktriangledown w))\blacktriangledown\,(v\blacktriangledown w) \quad \text{by prop.15} \\
&= (((w \,\mathrm{on}\, u)\blacktriangledown w) \wedge (v\blacktriangledown w))\blacktriangledown\,(v\blacktriangledown w) \quad \text{by prop.19} \\
&= (((w \,\mathrm{on}\, u) \wedge v)\blacktriangledown w)\blacktriangledown\,(v\blacktriangledown w) \quad \text{by prop.15} \\
&= (v \wedge (w \,\mathrm{on}\, u))\blacktriangledown\,(v \wedge w) \quad \text{by prop.25} \\
&= (w \,\mathrm{on}\, u)\blacktriangledown\,(v \wedge w) \quad \text{by prop.18}
\end{aligned}$$

□

Proposition 24. $\forall\, u \in \mathbb{B}^*,\ \forall\, v, w \in \mathbb{B}^{|u|_1}$,

$$(u \,\mathrm{on}\, v)\blacktriangledown\,(u \,\mathrm{on}\, w) = v\blacktriangledown w$$

Proof. We show per induction the correctness of the property. First, let suppose that $u = \varepsilon$. Then, $v = w = \varepsilon$ per definition of the *on* and *when* operators.

$$\varepsilon\blacktriangledown\varepsilon = (\varepsilon \,\mathrm{on}\, \varepsilon)\blacktriangledown\,(\varepsilon \,\mathrm{on}\, \varepsilon)$$

Now, we suppose the property true with $|u| = n - 1$ (u_{tail}), and we show its correctness with $|u| = n$. If $u_{head} = 0$, we have $|u_{tail}|_1 = |u|_1 = |v| = |w|$, and thus:

$$\begin{aligned}
(u \,\mathrm{on}\, v)\blacktriangledown\,(u \,\mathrm{on}\, w) &= (0.u_{tail} \,\mathrm{on}\, v)\blacktriangledown\,(0.u_{tail} \,\mathrm{on}\, w) \\
&= (0.\,(u_{tail} \,\mathrm{on}\, v))\blacktriangledown\,(0.\,(u_{tail} \,\mathrm{on}\, w)) \\
&= \varepsilon.\,((u_{tail} \,\mathrm{on}\, v)\blacktriangledown\,(u_{tail} \,\mathrm{on}\, w)) \\
&= v\blacktriangledown w
\end{aligned}$$

Else, if $u_{head} = 1$, we have $|u_{tail}|_1 = |u|_1 - 1 = |v_{tail}| = |w_{tail}|$, and thus:

$$\begin{aligned}
(u \,\mathrm{on}\, v)\blacktriangledown\,(u \,\mathrm{on}\, w) &= (1.u_{tail} \,\mathrm{on}\, v)\blacktriangledown\,(1.u_{tail} \,\mathrm{on}\, w) \\
&= (v_{head}\blacktriangledown w_{head}).\,((u_{tail} \,\mathrm{on}\, v_{tail})\blacktriangledown\,(u_{tail} \,\mathrm{on}\, w_{tail})) \\
&= (v_{head}\blacktriangledown w_{head}).\,(v_{tail}\blacktriangledown w_{tail}) \\
&= v\blacktriangledown w
\end{aligned}$$

Which concludes the proof. □

Proposition 25. $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v, w \in \mathbb{B}^n$,

$$u\blacktriangledown\,(v \wedge w) = (u\blacktriangledown w)\blacktriangledown\,(v\blacktriangledown w)$$

Proof.

$$\begin{aligned} u\blacktriangledown(v\wedge w) &= (u\wedge w)\blacktriangledown(v\wedge w) \quad \text{by prop.18}\\ &= (w\,\mathrm{on}\,(u\blacktriangledown w))\blacktriangledown(w\,\mathrm{on}\,(v\blacktriangledown w)) \quad \text{by prop.20}\\ &= (u\blacktriangledown w)\blacktriangledown(v\blacktriangledown w) \quad \text{by prop.24}\end{aligned}$$

□

Proposition 26. $\forall\, n\in\mathbb{N}^*,\ \forall\, u,w\in\mathbb{B}^n,\ \forall\, v\in\mathbb{B}^{|u|_1},$

$$(u\,\mathrm{on}\,v)\wedge w = (u\wedge w)\,\mathrm{on}\,(v\blacktriangledown(w\blacktriangledown u))$$

Proof.

$$\begin{aligned} (u\,\mathrm{on}\,v)\wedge w &= (u\,\mathrm{on}\,v)\wedge(u\wedge w) \quad \text{by prop.1 and 5}\\ &= (u\,\mathrm{on}\,v)\wedge(u\,\mathrm{on}\,(w\blacktriangledown u)) \quad \text{by prop.20}\\ &= u\,\mathrm{on}\,(v\wedge(w\blacktriangledown u)) \quad \text{by prop.9}\\ &= u\,\mathrm{on}\,((w\blacktriangledown u)\,\mathrm{on}\,(v\blacktriangledown(w\blacktriangledown u))) \quad \text{by prop.20}\\ &= (u\,\mathrm{on}\,(w\blacktriangledown u))\,\mathrm{on}\,(v\blacktriangledown(w\blacktriangledown u)) \quad \text{by prop.8}\\ &= (u\wedge w)\,\mathrm{on}\,(v\blacktriangledown(w\blacktriangledown u)) \quad \text{by prop.20}\end{aligned}$$

□

Proposition 27. $\forall\, n\in\mathbb{N}^*,\ \forall\, u,w\in\mathbb{B}^n,\ \forall\, v\in\mathbb{B}^{|u|_1},$

$$(u\,\mathrm{on}\,v)\blacktriangledown w = (u\blacktriangledown w)\,\mathrm{on}\,(v\blacktriangledown(w\blacktriangledown u))$$

Proof.

$$\begin{aligned} (u\,\mathrm{on}\,v)\blacktriangledown w &= ((u\,\mathrm{on}\,v)\wedge w)\blacktriangledown w \quad \text{by prop.18}\\ &= ((u\wedge w)\,\mathrm{on}\,(v\blacktriangledown(w\blacktriangledown u)))\blacktriangledown w \quad \text{by prop.26}\\ &= (u\blacktriangledown w)\,\mathrm{on}\,(v\blacktriangledown(w\blacktriangledown u)) \quad \text{by prop.22}\end{aligned}$$

□

Proposition 28. $\forall\, n\in\mathbb{N}^*,\ \forall\, u,v\in\mathbb{B}^n,\ \forall\, w\in\mathbb{B}^{|v|_1},$

$$u\blacktriangledown(v\,\mathrm{on}\,w) = (u\blacktriangledown v)\blacktriangledown w$$

Proof.

$$\begin{aligned} u\blacktriangledown(v\,\mathrm{on}\,w) &= u\blacktriangledown(v\wedge(v\,\mathrm{on}\,w))\\ &= (u\blacktriangledown v)\blacktriangledown((v\,\mathrm{on}\,w)\blacktriangledown v) \quad \text{by prop.25}\\ &= (u\blacktriangledown v)\blacktriangledown w \quad \text{by prop.19}\end{aligned}$$

□

Proposition 29. $\forall\, n \in \mathbb{N}^*,\ \forall\, u, v, w \in \mathbb{B}^n,$

$$(u\blacktriangledown w) = (v\blacktriangledown w) \Rightarrow u \wedge w = v \wedge w$$

Proof.

$$\begin{array}{rcll} (u\blacktriangledown w) = (v\blacktriangledown w) & \Rightarrow & w \operatorname{on} (u\blacktriangledown w) = w \operatorname{on} (v\blacktriangledown w) & \\ & \Rightarrow & u \wedge w = v \wedge w & \text{by prop.20} \end{array}$$

□

4.5 Sampling binary words

Definition 15 (Packed words). *A* n-packed word *is a binary word built over* $\{0^n, 1^n\}$*. In other word, a binary word is* n*-packed* iff*:*

- *its length is a multiple of* n*;*
- *when the word is parsed from its beginning, in subwords of length* n*, each subword is made only of "0"s (*exclusive-*)or "1"s.*

If more than one n *ensures this property, we just mention the biggest of them (which is the smallest common multiple among them). We assume that* ε *is 0-packed.*

Example 6. 10100 *is* 1*-packed.* 11110000 *is* $\{1, 2, 4\}$*-packed, so we will just call it* 4*-packed.*

Definition 16 (*When* operator with integer operand). *Let* w *be a* n*-packed word. The sampling of* w*, to gather just 1 bit of each subword of length* n*, is equivalent to apply a* when *operator, with* $\left(0^{n-1}.1\right)^{\frac{|w|}{n}}$ *as right operand. In order to shorten the notation, it will be further written* $w\blacktriangledown n$*, such that:*

$$w\blacktriangledown n = w\blacktriangledown \left(0^{n-1}.1\right)^{\frac{|w|}{n}}$$

Example 7.

$$11000011\blacktriangledown 2 = 1001$$

Definition 17 (*Pack* operator). *The pack operator (noted* ▲*) is recursively defined such as:*

$$\forall\, n \in \mathbb{N},\ \forall\, w \in \mathbb{B}^*,\ w\blacktriangle n = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ w^n_{head}.\,(w_{tail}\blacktriangle n) & \text{otherwise} \end{cases}$$

Note: if $w \neq \varepsilon$*,* $w\blacktriangle n$ *is at least* n*-packed.*

Example 8.

$$1001\blacktriangle 2 = 11000011$$

Definition 18 (Packed words equivalence). *Let u and v be two packed words. They are said to be equivalent* iff*:*

$$u \equiv v \;\Leftrightarrow\; \exists\,(m,n) \in \mathbb{N}^2,\; u\blacktriangle m = v\blacktriangle n$$

Basically, the idea behind this operators is to under- or over-sample binary words. Note that using the *when* operator in a such way with a non-packed word as left operand would be a non-sense, it would come to an under-sampling with loss of information.

5 Synchronous extension

Previously, we have introduced the KEG model in an asynchronous setting. Here we refine the previous model using an asap firing rule (in a synchronous way as described in [4]), we can now remove "unecessary" places in between *Select/Merge* (introducing instaneous links) and introduce a *copy* computation node in order to have an explicit mean to handle multiplicity of a token. Using the asap firing rule let us define the *throughput* and *schedule* of a KEG. We introduce the definition of the synchronous extension, we show how we can decide safety using also an abstraction on the SDF domain, finally we discuss throughput and schedule of such KEG.

5.1 Definition

Definition 19 (Synchronous extension). *A* synchronous KEG *is a structure $KEG = \langle \mathcal{N}, C, Se, Me, P, M_0, A\rangle$, together with several auxiliary functions listed below, such that:*

- *$\mathcal{N}$ is a finite set of computation nodes.*
- *C is a finite set of* copy nodes *such that $\forall\, c \in C : |c\bullet| \geq 1$ and $|\bullet\, c| = 1$.*
- *Se is a finite set of* Select *nodes such that $\forall\, se \in Se : |se\bullet| = 2$ and $|\bullet\, se| = 1$. We denote Se_{in} and Se_{out} the set of input/output of Se.*
- *Me is a finite set of* Merge *nodes such that $\forall\, me \in Me : |me\bullet| = 1$ and $|\bullet\, me| = 2$. We denote Me_{in} and Me_{out} the set of input/output of Me.*
- *P is a finite set of* places *such that $\forall\, p \in P : |p\bullet| = |\bullet\, p| = 1$ (i.e. there is only one consumer and only one producer:* conflict-free*).*
- *M_0 is the initial marking of places $M : P \to \mathbb{N}$.*
- *A is a finite set of* arcs *such that: $A \subseteq (\mathcal{N} \times (P \cup Se_{in} \cup C)) \cup (C \times (P \cup Se_{in})) \cup (P \times (\mathcal{N} \cup C \cup Me_{in})) \cup (Se_{out} \times (P \cup Se_{in})) \cup (Me_{out} \times (\mathcal{N} \cup Se_{in} \cup C))$*

  We add the following constraint in order to avoid combinatorial loop: there must exist at least a place in each directed cycle.

- *SwitchCond is a function* $Me \cup Se \longrightarrow \mathbb{P}$.

We still assume that $\forall f \in Me \cup Se$, $|SwitchCond|_1 = |SwitchCond|_0 = \infty$, so that switching conditions are fair.

Definition 20 (Firing rule). *We use the asap firing rule, when a Select, Merge, copy or computation node is enabled, then it fires.*

Now, we define a schedule and a throughput in the case of a strongly-connected KEG.

Definition 21 (Schedule). *For each Select, Merge, copy or computation node, we associate a k-periodic sequence, where each occurence of* 1 *denotes an execution, and* 0 *non execution.*

We assume that the given KEG is safe and live; we will show how to ensure for safety in subsection 5.2.

Definition 22 (Throughput). *The* throughput *of (copy-) computation node,* Select *or* Merge *is the* rate *of the schedule during its periodic part.*

However, the throughput of a KEG is not the minimum throughput among previous components as show on figure 8. The throughput of a strongly-connected KEG is throughput of the attached component on an input or output.

The throughput of an acyclic KEG is equal to one.

The throughput of a general graph is the list of throughput for each input/output.

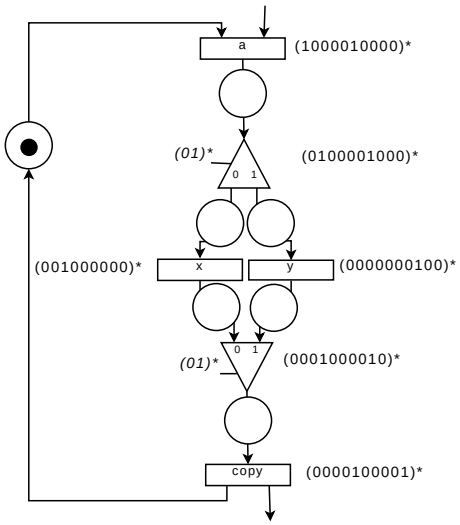

Figure 8: KEG Throughput

5.2 Abstraction into SDF

We now define an abstraction function of a KEG 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. KEG will then inherit the well-balanced conditions and safety properties from SDF theory.

Definition 23 (KEG abstraction to SDF). *Given a KEG we build its SDF abstraction* $\langle \mathcal{N}', P', M_0, W \rangle$ *as follows:*

- $\mathcal{N}' = \mathcal{N} \cup C \cup Se' \cup Me'$ *where* Se' *and* Me' *are two sets of computation nodes resulting from the abstraction of Selects and Merges.*
- $P' = P \cup P'_a$ *in which nodes in* Se' *and* Me' *have been substituted for nodes in* Se *and* Me*, and* P'_a *are additional places introduced when an arc is linking a pair of components such that both components are not places: that is to say that the link is "instantaneous". Such instantaneous links are described by the following set:*
  $Instantaneous = (\mathcal{N} \times (Se_{in} \cup C)) \cup (C \times Se_{in}) \cup (Se_{out} \cup Se_{in}) \cup (Me_{out} \times (C \cup Se_{in}))$.
  We have to introduce those places because in SDF there is only arcs from a computation node to a place, and from a place to a computation node. Introducing additional places do not change the "behaviour", because we are confluent*, computation node executions are just delayed in the transformed graph.*
- M_0 *is the initial marking of places.*
- W_{in}/W_{out} *are the weight functions associated for each place input and output respectively:* $\forall p' \in P'$

  $if\ source(p') \in (\mathcal{N} \cup C) : W_{in} = 1.$
  $if\ target(p') \in (\mathcal{N} \cup C) : W_{out} = 1.$
  $if\ source(p') \in Se' : W_{in} = |SwitchCond|.$
  $if\ target(p') \in Me' : W_{out} = |SwitchCond|.$
  $if\ \forall x \in [0,1] : target(p') \in Se'_x : W_{out} = |SwitchCond|_x.$
  $if\ \forall x \in [0,1] : source(p') \in Me'_x : W_{in} = |SwitchCond|_x.$

  The idea in the previous definitions is that, while plain $\mathcal{N}$ *and* C *just produce/consume one token on each channel, the production/consumption rate of* Select/Merge *can only be considered at the level of its switching pattern period.*

An example of abstraction from a KEG to SDF as already been shown previously in figure 5.

Property 2. *The synchronous extension of a KEG* K *is* safe *iff its SDF abstraction is.*

The proof follows the same scheme as in the abstraction from KEG to SDF.

5.3 Routing and scheduling

Using the operators *on* and *when* defined earlier, we can establish more precise properties on the scheduling inferred by our *Select* nodes, as well as compute the buffers sizes upstream the *Merge* nodes.

From now, we will denote $In \in \mathbb{P}$ the token flow (or *schedule*) on the input of the *Select* node, and $Out^0 \in \mathbb{P}$ (resp. Out^1) its left (resp. right) output flow. The opposite will be true for a *Merge* node.

Definition 24 (Substring). *We denote* $\text{substr}\,(Flow, begin, length) \in \mathbb{B}^*$ *where* $Flow \in \mathbb{B}^*$, $begin \in \mathbb{N}^*$ *and* $length \in \mathbb{N}$ *such that:*

$$\text{substr}\,(Flow, begin, length) = Flow_{begin}. \ \ldots \ .Flow_{begin+length}$$

5.3.1 Scheduling Select nodes

Proposition 30 (Flows on Select). *As shown in figure 9, the schedules on the Select outputs are:*

$$Out^0 = In \text{ on } \overline{c}$$
$$Out^1 = In \text{ on } c$$

where $c \in \mathbb{P}$ *is the switch condition.*

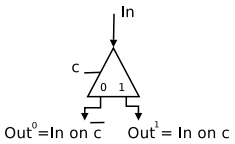

Figure 9: Schedules on Select node

Proof. The *on* operator definition originates from this property. Basically, for each token ("1") on the *Select* input, we look the corresponding bit in the switch condition c to determine whether it will be routed on the left or right output.

We can prove the token preservation as:

$$\begin{aligned} Out^0 \oplus Out^1 &= (In \text{ on } \overline{c}) \oplus (In \text{ on } c) \\ &= In \text{ on } (c \oplus \overline{c}) \\ &= In \text{ on } 1^{|c|} \\ &= In \end{aligned}$$

□

5.3.2 Scheduling Merge nodes

Proposition 31 (Flow on Merge).

$$\begin{array}{rl} \forall\, i \in [\![1, \left|In^0\right|_1 + \left|In^1\right|_1]\!], & Out_{\mathrm{t}(i)} = 1 \\ otherwise, & Out_{others} = 0 \end{array}$$

where t is the function defined as follows:

$$\mathrm{t} : \begin{cases} \mathbb{N} \longrightarrow \mathbb{N} \\ i \longmapsto \begin{cases} 0 & \text{if } i = 0 \\ \max\left(\left[In^{c_i}\right]_{|\mathrm{substr}(c,1,i)|_{c_i}}, \mathrm{t}\,(i-1)+1 \right) & otherwise \end{cases} \end{cases}$$

Proof. The function t gives us the date of outlet for i^{th} token of Out. To explain it part by part:

- $|\mathrm{substr}\,(x,1,i)|_1$ is the number of tokens passed in any flow x at instant i
- $\left[In^{c_i}\right]_{|\mathrm{substr}(c,1,i)|_{c_i}}$ is the position of "1" on inlet of the corresponding flow, for any "1" on outlet
- the max comes from the fact that earlier tokens may have been accumulated ; they will pass first

□

5.3.3 Maximum sizes of Merge buffers

Proposition 32.

$$max\_buf_{In^0} = \max_{i=1}^{|In^0|} \left(\left|\mathrm{substr}\left(In^0,1,i\right)\right|_1 - |\mathrm{substr}\,(c,1,\mathrm{awaiting}\,(i))|_0 \right)$$

$$max\_buf_{In^1} = \max_{i=1}^{|In^1|} \left(\left|\mathrm{substr}\left(In^1,1,i\right)\right|_1 - |\mathrm{substr}\,(c,0,\mathrm{awaiting}\,(i))|_1 \right)$$

where:

- $\left|\mathrm{substr}\left(In^0,1,i\right)\right|_1$ *is the number of tokens arrived on* In^0 *at time* i
- $\mathrm{awaiting} : i \longmapsto |\mathrm{substr}\,(Out,1,i)|_1 + 1 - Out_i$
- $|\mathrm{substr}\,(c,1,\mathrm{awaiting}\,(i))|_0$ *is the number of tokens already passed*

6 Transformations on Select and Merge

We now provide local transformations on *Merges* and *Selects*, which respect the behavior. We first define the order relation used to characterize the flow

equivalence, followed by the definitions of *on* and *when* operators, applying on binary words, which will be used to transform the switching conditions.

As we will see in this section, 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 then be forbidden by excluding schedules that would display them). Using previous token-flow relation, we show how we can mix together routing and scheduling in order to computer the size of required buffers. After, we describe local transformations that are preserving token-flows. Finally, we detail the expansion process, which is unsharing communication links. 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. **toujours d'actualité la dernière phrase ?**

6.1 Order relation

Definition 25 (Elementary token-flow relation). *For each node (resp. place), we associate to each token of the input flow a couple* $(i, j) \in \mathbb{N}^* \times \mathbb{N}^*$*, where* i *represents the token passing index on the node (resp. place) input, and* j *represents the token passing index on the node (resp. place) output.*

There exists a set of firing relations *of the form* $\curvearrowright \subseteq (\mathbb{N}^* \times \mathbb{N}^*)$*, such that for the concerned node (resp. place),* $i \curvearrowright j$*. They are defined as follows:*

- $\forall\, p \in P, \forall\, i \in \mathbb{N}^*,\ i \curvearrowright_p (i + M_0(p))$*, where* $M_0(p)$ *is the initial marking of* p
- $\forall\, n \in \mathcal{N}, \forall\, i \in \mathbb{N}^*,\ i \curvearrowright_n i$
- $\forall\, s \in Se, \forall\, i \in \mathbb{N}^*,\ [\overline{SwitchCond(s)}]_i \curvearrowright_{s,0} i$
- $\forall\, s \in Se, \forall\, i \in \mathbb{N}^*,\ [SwitchCond(s)]_i \curvearrowright_{s,1} i$
- $\forall\, m \in Me, \forall\, i \in \mathbb{N}^*,\ i \curvearrowright_{m,0} [\overline{SwitchCond(m)}]_i$
- $\forall\, m \in Me, \forall\, i \in \mathbb{N}^*,\ i \curvearrowright_{m,1} [SwitchCond(m)]_i$

These relations are monotone*, that is to say:*

$$\forall\ (i_1, j_1), (i_2, j_2) \in \curvearrowright,\ i_1 \leq i_2 \Leftrightarrow j_1 \leq j_2$$

Places and computation nodes do not alter ordering of tokens. With *Selects* and *Merges*, the relations between input and output flows depend on switching conditions; we have to consider if the token goes through the left or right output (resp. input).

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

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.

Definition 26 (Token-flow relation over a path). *Let Σ be the set of all simple paths in a KEG. A path $\sigma \in \Sigma$ is of the form $\sigma = n_1.p_1.n_1. \ldots .n_{l+1}$, with $\forall\, i \in [\![1, l+1]\!],\ n_i \in \mathcal{N} \cup Se \cup Me$ and $\forall\, j \in [\![1, l]\!],\ p_j \in P$.*

We associate to each path σ a relation $\curvearrowright_\sigma \subseteq (\mathbb{N}^* \times \mathbb{N}^*)$ such that $\curvearrowright_\sigma = \curvearrowright_{n_1} \circ \curvearrowright_{p_1} \circ \ldots \circ \curvearrowright_{n_{l+1}}$, where $\curvearrowright_a \circ \curvearrowright_b = \{(i,k) \mid \forall\, (i,j) \in a,\ (j,k) \in b\}$.

Being the composite of monotone relations, $\curvearrowright_\sigma$ is a monotone relation.

Definition 27 (Order preservation). *We call a KEG E* order-preserving *iff $\forall\, n, n' \in \mathcal{N},\ \forall\, \sigma_1, \sigma_2 \in \Sigma_{\sigma_1 \to \sigma_2} \,/\, \sigma_1 \neq \sigma_2,\ \curvearrowright_{\sigma_1} \cup \curvearrowright_{\sigma_2}$ is monotone.*

In an order-preserving KEG, for each path between two computation nodes, the tokens emitted by the source will arrive to the sink in the same order.

6.2 Local transformations

We introduce now a set of local transformations on a KEG, we show correctness of those transformations using order preservation of tokens and previous properties.

Definition 28. *We consider the three following graph transformations:* Permuting Merges*,* Permuting Selects*, and* De-sharing *with the three associated figures 10, 11, and 12 respectively, where a, b, c, d are tokens flows and $u, v \in \mathbb{P}$ are switching conditions.*

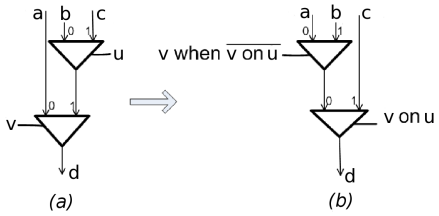

Figure 10: Permuting *Merges*

Transformation *Permuting Merges* (resp. *Permuting Selects*) isolates an input (resp. output) from a shared channel to a dedicated one. The transformation *De-sharing* consists in splitting a shared channel, as displayed in figure 12. If such channel is shared (as in figure 12a), then tokens may stack up with flows a, b, awaiting for the other side to take its turn. When the channels are split, more concurrency is allowed.

It should also be noticed, concerning transformation of figure 12, that it is not reversible in general. As depicted in figure 13, which shows possible

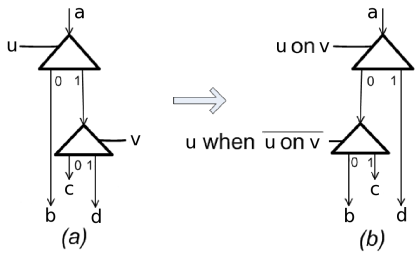

Figure 11: Permuting *Selects*

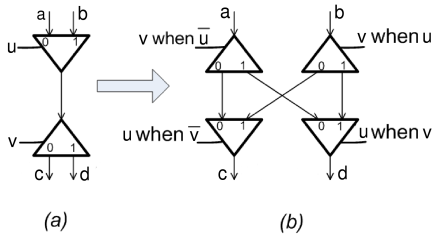

Figure 12: "De-sharing" an edge

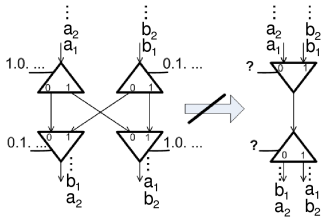

Figure 13: Sharing not always feasible

behaviors on flow prefixes of length 2, the shared version cannot modify the token order as in the unshared version.

Property 3. *The transformations depicted in figures 10, 11 and 12, with the corresponding relevant transformations on switching conditions, preserve the token flows.*

Proof. We show the correctness of the transformation depicted in figure 10, using both token count and order preservation on flows a, b, c and d. Here, it is obvious that the number of tokens of d is invariant.

Using definitions 25 and 26, we have:

$$\begin{array}{rcl} \textit{Figure } 10\ (a): & & \textit{Figure } 10\ (b): \\ a \curvearrowright_{m,0} [\overline{v}]_a & & a \curvearrowright_{m,0} \circ \curvearrowright_{m,0} [\overline{v \operatorname{on} u}]_{[v \blacktriangledown \overline{v \operatorname{on} u}]_a} \\ b \curvearrowright_{m,0} \circ \curvearrowright_{m,1} [v]_{[\overline{u}]_b} & \leftrightarrow & b \curvearrowright_{m,1} \circ \curvearrowright_{m,0} [\overline{v \operatorname{on} u}]_{[v \blacktriangledown \overline{v \operatorname{on} u}]_b} \\ c \curvearrowright_{m,1} \circ \curvearrowright_{m,1} [v]_{[u]_c} & & c \curvearrowright_{m,1} [v \operatorname{on} u]_c \end{array}$$

Then, we have by corollary 3 and properties 10 and 20:

$$\begin{aligned} \overline{v \operatorname{on} u} \operatorname{on} \overline{v \blacktriangledown \overline{v \operatorname{on} u}} &= (v \operatorname{on} u) \oplus \overline{\overline{v \operatorname{on} u} \operatorname{on} (v \blacktriangledown \overline{v \operatorname{on} u})} \\ &= (v \operatorname{on} u) \oplus \overline{v \wedge \overline{v \operatorname{on} u}} \\ &= (v \operatorname{on} u) \oplus (\overline{v} \vee (v \operatorname{on} u)) \\ &= \overline{v} \end{aligned}$$

$$\begin{aligned} \overline{v \operatorname{on} u} \operatorname{on} (v \blacktriangledown \overline{v \operatorname{on} u}) &= v \wedge \overline{v \operatorname{on} u} \\ &= v \wedge (\overline{v} \oplus (v \operatorname{on} \overline{u})) \\ &= v \operatorname{on} \overline{u} \end{aligned}$$

Finally, using property 3, we show for each path starting from input a, b or c, that in both cases, the output orders are the same.

The same reasoning can be applied to prove the transformations of figures 11 and 12. □

Property 4. *The transformations displayed in figures 10, 11, and 12 do not alter the safety of the graph.*

Proof. According to property 3, the transformations preserve token-flow so if the SDF abstraction of the original graph is safe, the SDF abstraction of the rewritten graph also is. □

Figure 14 presents a pattern to alter order among tokens.

Several things are worth noticing on that simple pattern. First of course u and v must have the same rate, otherwise tokens may accumulate in one of the inner channels. If the switching conditions of the upfront *Select* and of the trailing *Merge* will exactly match, so that the tokens exit in the same order that

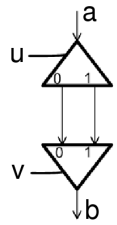

Figure 14: Simple pattern to permute tokens

they enter it. In this ideal case the graph can altogether be replaced by a single place channel.

Conversely, a sequence of this simple pattern may be used to replace a single channel to manage out-of-order computations in a predictable way (according to the switching conditions). The tokens entering one channel may be bypassed by tokens coming from the other channel. For instance, if $u = (011)^\omega$ and $v = (110)^\omega$, then every third token is kept in the "0-side" place channel to be bypassed by its two next followers along the "1-side".

6.3 Expansion process

The expansion is an algorithm, that is "unsharing" links. The intuitive idea have already been introduced in figure 12. We describe here a generalization for this one, in figure 15, there is a subset *me* of Merges (in the figure from 1 to n) and only one Select that is *dominator* (that is to say that all flows coming from the subset are passing through this *Select se*) and the *Select* receive only those flows. Then, we can duplicate $2^{|me|*2}$ times the inner DAGs, compute associate conditions for switching patterns and eliminate dead-code using schedule conditions.

For instance, two or more functionnaly equivalent Strongly Connected Components (*SCC*s) of the KEG may have been factorized, using *Selects* and *Merges* nodes ; a parallel computation is transformed this way in a sequential computation. Thus, the conditions of the introduced *Selects* and *Merges* are particular binary words, formed by the concatenation of n-ary subwords which contain only "0" (*exclusive-*)or "1" bits (that is to say packed words), n being the number of tokens produced or consumed during a full computation.

Expansion algorithm We assume that the input KEG is *safe*.

- We find all Strongly Connected Components (SCCs) in the KEG. We abstract each SCC as a computation node, and we obtain a new acyclic KEG. (Complexity: $O(V + E)$.)
- For each "input" SCC, that is to say a SCC which is not having any input: we change the label for each target computation node adding the

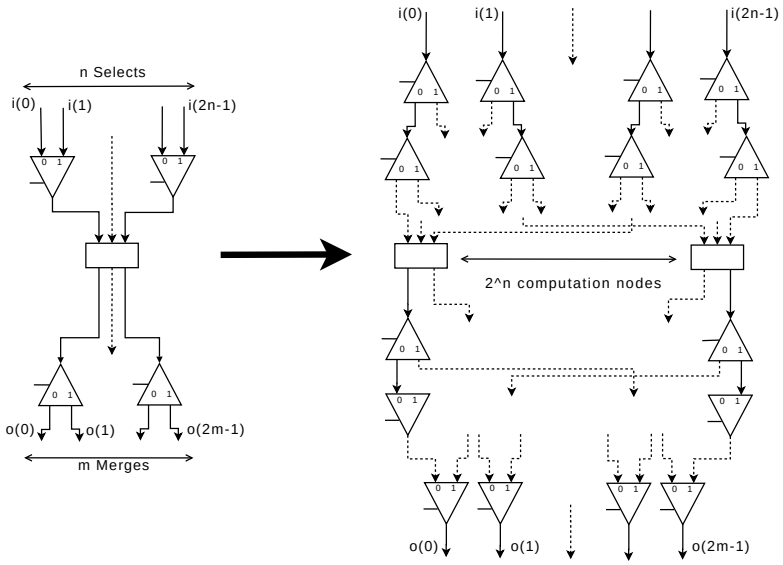

Figure 15: Expansion of shared DAG of SCCs

SCC name, then we remove the computation node and associated link for this SCC.

- Starting from inputs in the acyclic KEG, we find a partial order for each node, using a longest path algorithm. (Complexity: $O(V' + E')$ where $|V'| < |V|$ and $|E'| < |E|$).
- For each me_i in the set $Me' = Me \backslash SCC$ (Merge), we check for each se_j in the set $Se' = Me \backslash SCC$ (Select) if the given *Select* is a "dominator", if so then we attach me_i to se_j. (Complexity: $O(|Me'| * |Se'|)$).
- (a) For each subset me_i in the set Me' (*Merge*) of size 1 to $|Me'|$, we check for each se_j in the set Se' (*Select*) if se_j is "dominator" and receive **only** the combined flows from me_i. (Complexity: $O(2^{|Me'|} * |Se'|)$)
  - We might have differents candidates, that is why we have initially partially ordered nodes. We take the candidate with the lowest order. (Complexity: $O(Candidates)$)
  - We are constructing the sub-graph from *Merges* inputs until reaching the candidate. We abstract the sub-graph as a computation node.
    - Then, we duplicate the sub-graph.
    - After, we compute the switching conditions for the expanded graph. We obtain a forest of *Selects* on inputs reaching all duplicated computation node (sub-graph), for each output compu-

tation node, we have a *Select* connected to a forest of *Merge* outputs as shown on figure 15.

* We apply dead-code elimination using the switching conditions: for each leaf of the obtained *Select* tree, if there is only 0 or 1 in the condition, then we can remove this link with sub-graph instance, since there is no production.
* Finally, we cleanup references in all *Selects* dominator list about the *Merges* in subset me_i, we cleanup candidate list, we recursively call (a) on the modified KEG.

Construction of the input network We are actually building a *selection forest*, where the leafs are the 2^{inputs} duplicated subgraph. The roots of the forest are the original inputs followed by a *Select* where its switching condition is parametrized by the original *Merge* condition, at each stage of the forest we add more constraints on the *Select* switching condition with respect to previous stages, until covering the 2^{inputs} cases and so reaching the 2^{inputs} duplicated sub-graphs.

Algorithm 1 buildUpperTrees(NUM_INPUTS)

for $n = 1$ to NUM_INPUTS **do**
  add buildInputTree($1, NUM\_INPUTS, n, \{\overline{M_n}\}$)
  add buildInputTree($1, NUM\_INPUTS, n, \{M_n\}$)
end for

Algorithm 2 buildInputTree($curDepth, maxDepth, n, cond$)

if $curDepth = maxDepth$ **then**
  return new ComputationNode($cond$)
else if $curDepth = n$ **then**
  return buildInputTree($curDepth + 1, maxDepth, n, cond$)
else
  return new Select(
    $M_{curDepth}\blacktriangledown\,(\bigwedge cond)\,\blacktriangle i_n$,
    buildInputTree($curDepth + 1, maxDepth, n, cond \cup \{\overline{M_{curDepth}}\}$),
    buildInputTree($curDepth + 1, maxDepth, n, cond \cup \{M_{curDepth}\}$)
)
end if

Construction of outputs networks There is dispatching network for each output *Select* in the original graph. Starting from the *mapped* output *Select* on each 2^{inputs} duplicated sub-graphs, we are reaching a *Select* where its condition is constrained by the input flow associated with the duplicated graph, and the original switching condition of the original output *Select*. After, we

connect each output of the *Select* to a *Merge forest*, that is collecting all the cases until reaching the two original outputs.

Algorithm 3 buildLastSelectsRow($NUM\_OUTPUTS$)

for all $t \in ComputationNodes$ **do**
  for $n = 1$ to $NUM\_OUTPUTS$ **do**
    link the n^{th} output of t with new Select($S_n\blacktriangledown\,(\bigwedge t.cond)\,\blacktriangle o_n, null, null$)
  end for
end for

Algorithm 4 buildLowerTrees($NUM\_OUTPUTS$)

for $n = 1$ to $NUM\_OUTPUTS$ **do**
  add buildOutputTree($1, NUM\_OUTPUTS, n, \{\overline{S_n}\}$)
  add buildOutputTree($1, NUM\_OUTPUTS, n, \{S_n\}$)
end for

Algorithm 5 buildOutputTree($curDepth, maxDepth, n, cond$)

if $curDepth = maxDepth$ **then**
  if $\neg S_n \in cond \wedge \neg M_{maxDepth} \in cond$ **then**
    return link with left child of Select s.t. $Select.cond = cond \setminus \{\overline{S_n}\}$
  else if $\neg S_n \in cond \wedge M_{maxDepth} \in cond$ **then**
    return link with right child of Select s.t. $Select.cond = cond \setminus \{\overline{S_n}\}$
  else if $S_n \in cond \wedge \neg M_{maxDepth} \in cond$ **then**
    return link with left child of Select s.t. $Select.cond = cond \setminus \{S_n\}$
  else $\{S_n \in cond \wedge M_{maxDepth} \in cond\}$
    return link with right child of Select s.t. $Select.cond = cond \setminus \{S_n\}$
  end if
else
  return new Merge(
    $M_{curDepth}\blacktriangledown\,(\bigwedge cond)\,\blacktriangle o_n$,
    buildOutputTree($curDepth + 1, maxDepth, n, cond \cup \{\overline{M_{curDepth}}\}$),
    buildOutputTree($curDepth + 1, maxDepth, n, cond \cup \{M_{curDepth}\}$)
)
end if

7 Example

In this section we discuss the implementation of an example, which is the well-known Sobel algorithm used for edge detection in an image. This algorithm is written as a nested-loop.

The figure 16 shows an example of source image, and after applying the Sobel filter.

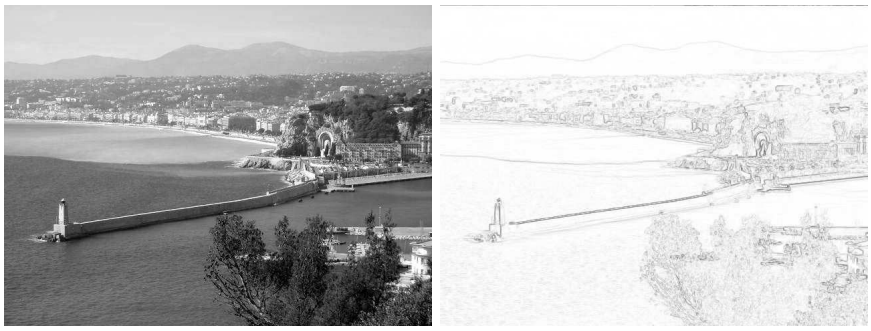

Figure 16: Input image and inverted Sobel result image

7.1 Sobel initialization

```
GX = { {+1, 0 , -1},{+2, 0 , -2},{+1, 0 , -1} }; //3 by 3 const
GY = { {+1, +2, +1},{ 0, 0 ,  0},{-1, -2, -1} }; //3 by 3 const

for(int X=0; X<=rows-1; ++X) { //filling part of result image
 edgeImage.data[X][0] = 0;
 edgeImage.data[X][cols-1] = 0;
}

for(int Y=0; Y<=cols-1; ++X) { //another part of result image
 edgeImage.data[0][Y] = 0;
 edgeImage.data[rows-1][Y] = 0;
}
```

7.2 Sobel main loop

```
for(int X=1; X<(rows-1); X++) { //scanning rows of source image
 for(int Y=1; Y<(cols-1); Y++) {//scanning columns of source image
   //BEGIN KERNEL LOOP transformed as a KEG
   int sumX = 0; int sumY = 0;

   for(int I=-1; I<=1; I++) {
     for(int J=-1; J<=1; J++) {
       sumX += originalImage[X+I][Y+J] * GX[I+1][J+1];
       sumY += originalImage[X+I][Y+J] * GY[I+1][J+1];
     }
   }

   int SUM = abs(sumX) + abs(sumY);

   //To 8 bits grey levels
```

```
   //data dependent control abstracted as dataflow
   if(SUM>255) { SUM=255; }

   edgeImage[X][Y] = 255 - SUM;
   //END KERNEL LOOP transformed as a KEG
 }
}
```

We will focus on the kernel inside the kernel loop to create a KEG.

7.3 From loopnest to KEG

First, we have to transform the loopnest in order to remove any data dependant control. In the case of the Sobel main loop, we have a data dependent $if - then - else$, we abstract it as only one instruction/computation node (that we call "8 bit grey"). Second, we can apply loop collapsing since the loop nest is perfect.

We start to describe our KEG implementation by the most easy part: which is the subgraph following the collapsed loop, as shown in the figure 17 (a). The transformation works as follows: for each instruction with associate a computation node. We have also a constant 255, to translate it in a KEG, we use a directed cycle with a copy computation node C_T to "recycle" the constant represented by an initial token.

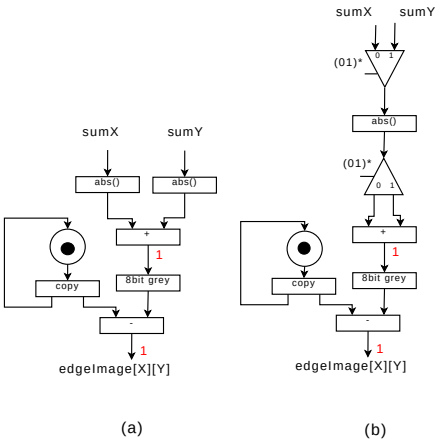

Figure 17: Before and after factorizing "abs()" nodes

Now we can focus on the most difficult parts which are loops (collapsed loop). When looking at dependencies in the inner most loops, we have two instructions that are parallel (multiply and accumulate (mac)). Both inner loops, have a domain on length 3. We will call the two macs 9 times, before living both loop.

First, we can note that $original[X+I][Y+J]$ has a multiplicity of 2 inside the loop. We use a copy computation node.

Second, it appears that both macs instructions can be executed in parallel when looking at dependence graph.

A loop is composed of :

- An optional initialization part followed by
- A synchronization start computation node whose inputs are necessary dependencies to start inner instructions computation, this computation node is linked to
- The control loop part, containing the instructions inside the loop.
- After the control loop part, we have the reset of the loop, and outputs of data.

Actually the initialization part is optional, and might contain in general constants that we translate using the previous transformation. In our example we have $sumX = 0, sumY = 0$, we use the same pattern as before for constant instruction: a simple place with a copy computation node that is recycling its value.

We connect on the other output of this copy a place to the synchronization start computation node, this computation node is having a place on its inputs with an initial token. When this computation node is fired, then the loop starts executing inner statement.

First, we take the constant value, then we compute the multiply, then the add which is sent back (accumulate) during 8 times, the last one is sent to a copy computation node, that is resetting the loop and sending the only result read from the loop in this case $sumX$ (and duplicating the pattern we obtain $sumY$). Next, we just have to forward both results outside of the loop.

7.4 Simple factorization on KEG

Consider figure 18 (b), there is two *abs* computation node in parallel, we can factorize this introducing a *Merge* before that is taking $sumX$, $sumY$ and just after applying *abs*, then we put a *Select* on the output of *abs* reaching finally the adder. As shown on figure 17.

Now we consider the inner kernel loop. There is two multiply and accumulate in the most inner loop nest, which applies on the same part of the image. We will use a single multiply and accumulate element. We alternate calculating the Sobel gradient on X, and next time on Y.

To do so, we need to get the 9 elements from GX, then 9 elements from GY as done by the *Merge* with the schedule $1^9.0^9$ where 1 input is connected to GX and 0 to GY as shown on the upper left corner of figure 19. We do also the same on the image input, $originalImage[X+I][Y+J]$ is used two times (copy computation node): one for the X gradient, and one for the Y gradient. We put a *Merge* just after the *copy* computation node, we copy the 9 elements

while there are passing through the *Merge* for computation of X, then we can compute Y. When reaching the outside we have only one output instead of two, we need to put a *Select* which is alternatively sending $sumX$,$sumY$. Hopefully, the output of this *Select* are connected to a *Merge* having exactly the same pattern, we can replace this pair of *Select/Merge* with a simple wire (but we do not apply this to check for canonical form and expansion algorithm).

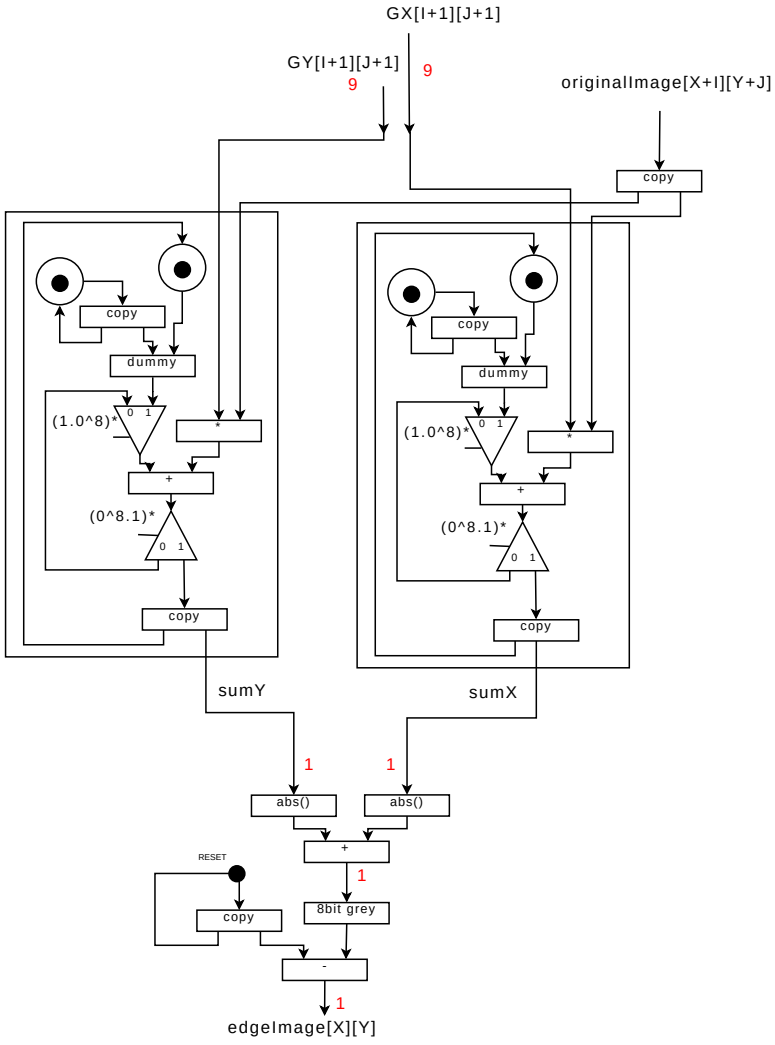

Figure 18: a KEG implementation of the Sobel filter

The main correctness criterion is that the factorized KEG implement a partial order of events which is compatible with the initial one. We argued that

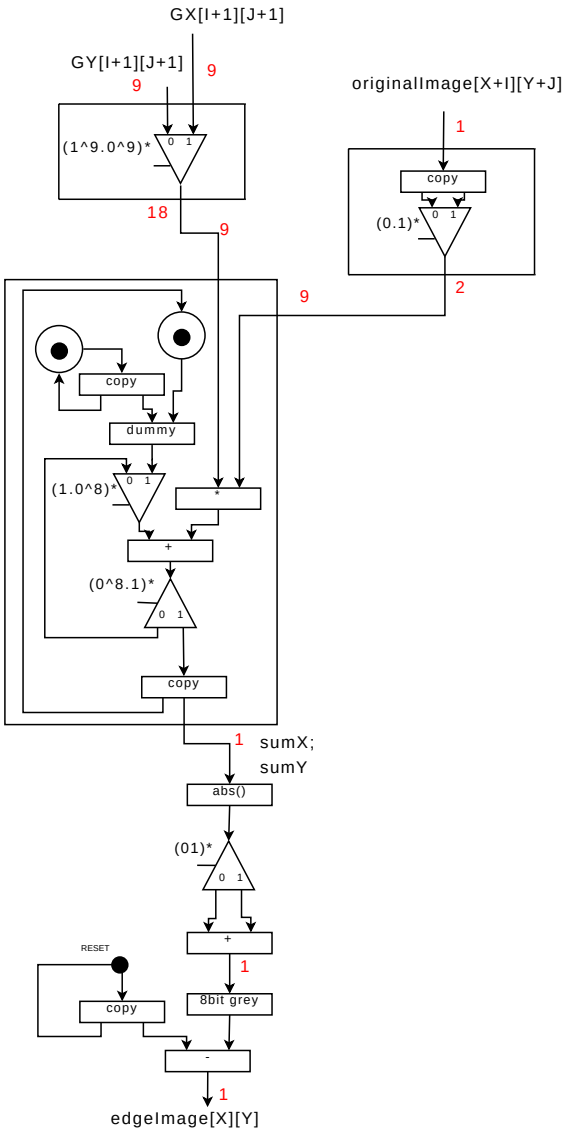

Figure 19: another KEG implementation of the Sobel filter

through an "expansion" process we are able to show that we have the same implementation in the factorized one than in the implementation without removing unecessary *Select/Merge* pairs.

We apply the "expansion":

1. We take the factorized KEG graph and search for strongly connected components (SCCs), we abstract each SCC as a computation node. In the

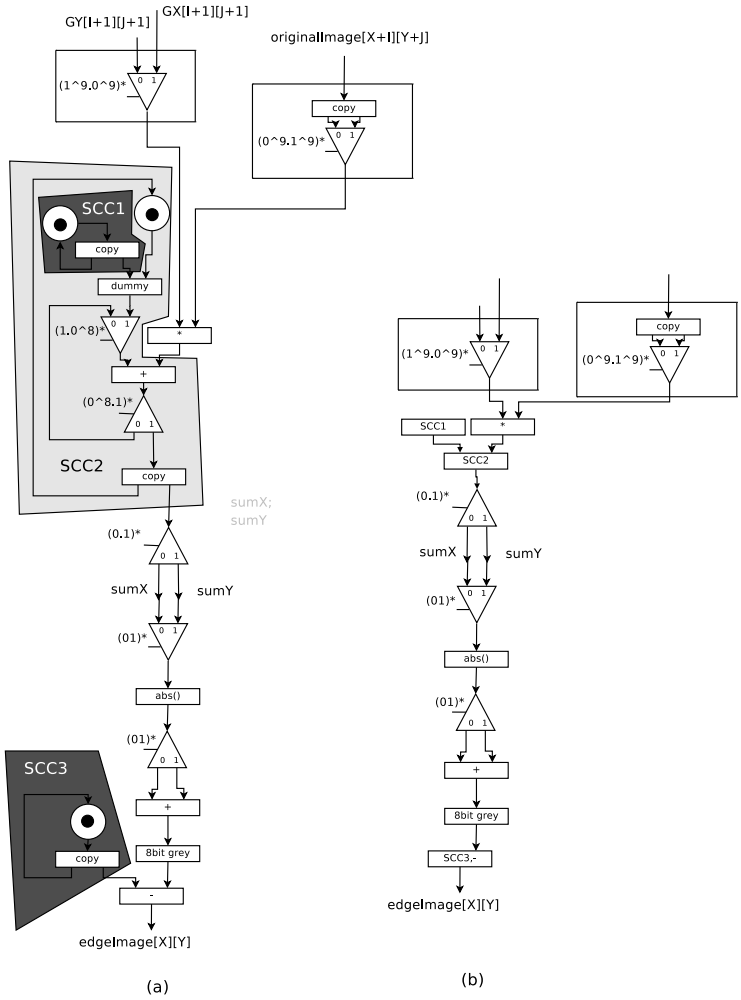

Figure 20: Expansion: (a) original KEG; (b) finding SCCs and "removing" input computation node

example, we have two SCCs one for the constant pattern in the loop and the loop nest itself. The constant pattern SCC, is an "input" computation node and can be safely merged in the SCC of the loop nest. Finally, we only have one SCC having one input that is the result of the multiplication and one output reaching a *Select*. As shown on figure 20 (a) and (b) respectively.

2. We partially order each node starting from inputs to outputs as shown on figure 21.
3. For each *Merge*, for each *Select*, check if the *Select* is a dominator for the given *Merge*, if so add this *Merge* to the dominator list of the given *Select* as shown on figure 21. For instance $5 : 1, 2, 5$ dominates 1 and 2.
4. Now, we are applying the main part of the "expansion":
   (a) For each *Merge*, we are looking if there is a *Select* which is a dominator and its flow is strictly equal to the one of the given *Merge*. Actually there is no one.
   (b) For each pair of *Merge*, there is only one *Select* candidate; which is 5 for pair $(1, 2)$ as shown on figure 21.
      i. Now, we are constructing the sub-graph linking this set of *Merge* to the candidate *Select* and abstract the whole as a computation node as shown on figure 21.
      ii. Then, we apply the expansion and obtain the figure 22 with associated additional *Select/Merge* with there conditions.
      iii. After, we apply the dead-code removal scanning conditions on added *Select/Merge* and remove uncessary nodes, we obtain then the figure 23.

Starting from the inputs, we are reaching two *Merges*. We can duplicate the sub-graph starting from output of the *Merge* until reaching the first *dominator Select* (all pathes from found inputs coming to this *Select*). Which is in this case just after the copy computation node. Then we go down in the graph until found a *Merge*, and so on.

8 Conclusion

This paper introduces the *Kahn-extended Event Graphs* model, which is adding alternating routes in the Marked/Event Graphs (MEG), through the introduction of *Select* and *Merge* nodes borrowed from Boolean-controlled Dataflow (BDF); *Select/Merge* are annotated with k-periodic binary words inspired from Cyclo-Static Data Flow (CSDF). Those words are independent of data values just like in Kahn Process Networks, moreover they are known at compile time. This model is confluent, concurrent, deterministic, with explicit communications through *Select/Merge* nodes and can be checked for safety (bounded buffering resources) by an abstraction in a Synchronous Data Flow graph. Through the use of an asap firing rule for any live and safe KEG, we can compute a static schedule and size of needed buffering resources. We can also add fixed latencies on both computation and communication in our model. We also introduce *on* and *when* operators applying on conditions of *Select/Merge*. Those operators enable the description of different transformations such as sharing and unsharing communication channels. Those transformations are preserving the behavior,

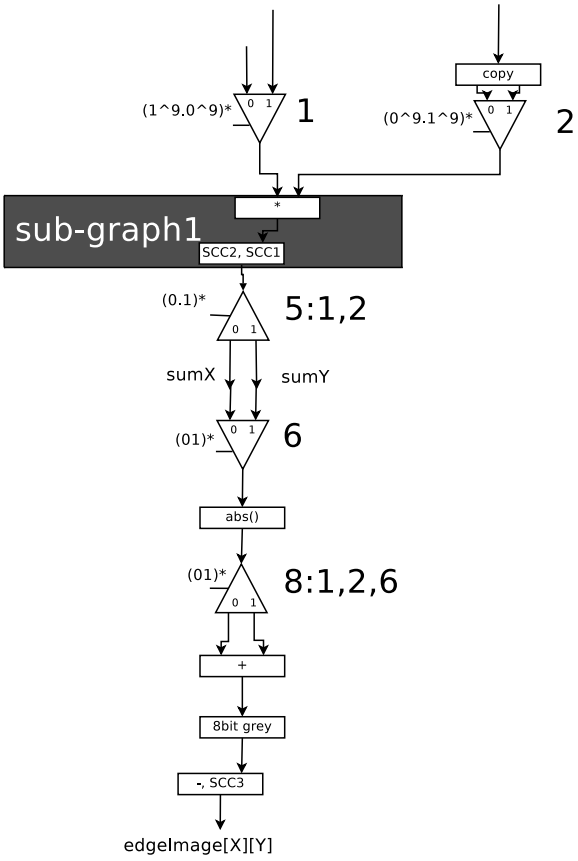

Figure 21: Construction of sub-graph between *Merge/Select*

shown through a flow-equivalence. We introduce the *expansion* algorithm, which is able to transform in some cases a sequential program in an equivalent more parallel one, while preserving its global behavior. We can also mix in the same framework latencies, schedules, switching conditions for optimization purpose.

Future work We think that this low-level (akin to assembly) model is a good candidate for code generation of *loop-nests* using the polyhedral model [21] and transformations on top of it. For instance, in Compaan [15], the linearization step introduce more sequentiality than strictly needed. Our model is able to handle both concurrency/communication which are mandatory for SoC, NoC, SiP and high performance computing, while having the previous desirable properties.

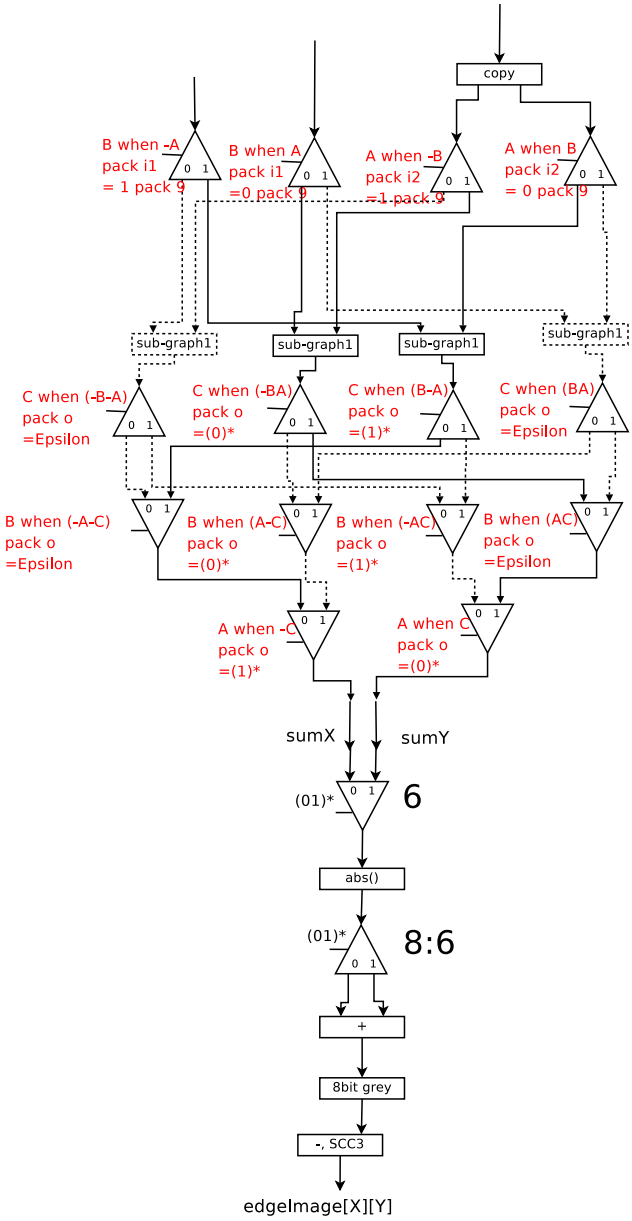

Figure 22: Expansion of the sub-graph

Those transformations (in particular the expansion) on Select/Merge might let us find some canonical forms of a KEG graph. Canonical forms are from utter importance, and can enable formal verification on this model.

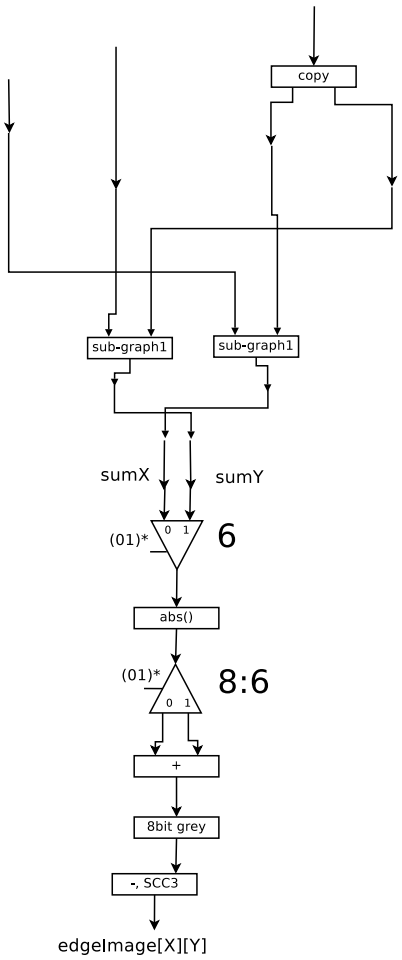

Figure 23: Dead-code elemination with *on/when/pack* operators

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Charles André, Frédéric Mallet, and Robert De Simone. Modeling time(s) in uml. Technical report, I3S - CNRS - UNSA, may 2007.

[3] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. In *IEEE Transactions on Signal Processing*, volume 4, February 1996.

[4] Julien Boucaron, Robert de Simone, and Jean-Vivien Millo. Latency-insensitive design and central repetitive scheduling. In *IEEE-ACM International Conference MEMOCODE'06*, pages 175–183. IEEE Press, 2006.

[5] Julien Boucaron, Robert de Simone, and Jean-Vivien Millo. Formal methods for scheduling of latency-insensitive designs. *EURASIP Journal on Embedded Systems*, 2007:Article ID 39161, 16 pages, 2007. doi:10.1155/2007/39161.

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

[7] Noam Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

[8] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks. In *POPL 2006 Proceedings*, January 2006.

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

[10] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Behavior and communication co-optimization for systems with sequential communication media. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 675–678, New York, NY, USA, 2006. ACM.

[11] Hartmann Genrich. Das zollstationenproblem. Technical Report GMD-I5/69-01-15, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, 1969.

[12] John E. Hopcroft and Jeffrey D. Ullman. *Formal languages and their relation to automata*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1969.

[13] Gilles Kahn. The semantics of a simple language for parallel programming. In *Inform. Process. 74: Proc. IFIP Congr. 74*, pages 471–475, 1974.

[14] Vinod Kathail, Shail Aditya, Robert Schreiber, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman. Pico: automatically designing custom computers. *Computer*, 35(9):39–47, September 2002.

[15] Bart Kienhuis, Edwin Rijpkema, and Ed Deprettere. Compaan: deriving process networks from matlab for embedded signal processing architectures. *Hardware/Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on*, pages 13–17, 2000.

[16] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10–23, May-June 2006.

[17] 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.

[18] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, 75(9):1235–1245, 1987.

[19] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[20] Carl A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962. Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65–377, Vol.1, 1966, Pages: Suppl. 1, English translation.

[21] Sébastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, Ottawa, Canada, June 2006.

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

[23] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

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

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

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