Network Fusion*

Pascal Fradet[1] and Stéphane Hong Tuan Ha[2]

[1] INRIA Rhône-Alpes
655, av. de l'Europe, 38330 Montbonnot, France
Pascal.Fradet@inria.fr

[2] IRISA/INRIA Rennes
Campus de Beaulieu, 35042 Rennes, France
Stephane.Hong_Tuan_Ha@irisa.fr

Abstract. Modular programming enjoys many well-known advantages but the composition of modular units may also lead to inefficient programs. In this paper, we propose an invasive composition method which strives to reconcile modularity and efficiency. Our technique, network fusion, automatically merges networks of interacting components into equivalent sequential programs. We provide the user with an expressive language to specify scheduling constraints which can be taken into account during network fusion. Fusion allows to replace internal communications by assignments and alleviates most time overhead. We present our approach in a generic and unified framework based on labeled transition systems, static analysis and transformation techniques.

1 Introduction

Modular programming enjoys many well-known advantages: readability, maintainability, separate development and compilation. However, the composition of modular units (components) gives rise to efficiency issues. Sequential composition poses space problems: the producer delivers its complete output before the consumer starts. Parallel composition relies on threads, synchronization and context switches which introduce time overhead.

In this paper, we propose an invasive composition method, *network fusion*, which strives to reconcile modularity and efficiency. Our technique automatically merges networks of interacting components into equivalent sequential programs. Our approach takes two source inputs: a network of components and user-defined scheduling constraints. Networks are formalized as *Kahn Process Networks* (KPNs) [7] a simple formal model expressive enough to specify component programming and assembly. Scheduling constraints allow the user to choose the scheduling strategy by specifying a set of desired executions. The operational semantics of KPNs and scheduling constraints are both formalized as guarded labeled transition systems (LTS).

* This work has been supported in part by the ACI DISPO and Région Bretagne

Network fusion is an automatic process which takes a Kpn and scheduling constraints and yields a sequential program respecting the constraints. Note that constraints may introduce artificial deadlocks, in which case the user will be warned. The resulting program must be functionally equivalent to the Kpn modulo the possible deadlocks introduced by constraints. Fusion alleviates most time overhead by allowing the suppression of context switches, the replacement of internal communications by assignments to local variables and optimizations of the resulting sequential code using standard compiling techniques. Network fusion can be seen as a generalization of *filter fusion* [12] to general networks using ideas from aspect-oriented programming [8] (scheduling constraints can be seen as an aspect and their enforcement as weaving).

The four main steps of the fusion process are represented in Figure 1.

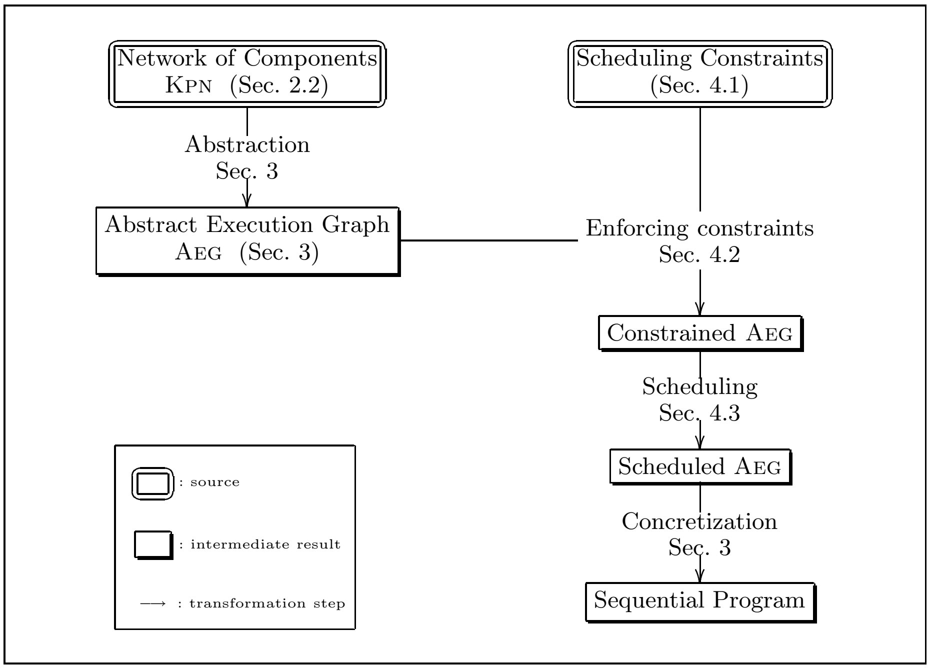


Fig. 1. Main steps of network fusion

- The first step is the *abstraction* of the network into a finite model called an *Abstract Execution Graph* (Aeg). An Aeg over-approximates the set of possible executions traces. We do not present this step in details since it relies on very standard analysis techniques (*e.g.,* abstract interpretation) and many different abstractions are possible depending on the desired level of precision. Instead, we focus on the properties that an Aeg must satisfy.

– The second step consists in *enforcing constraints*. This is expressed as a synchronized product between guarded LTS (the AEG and the constraints). In general, this step does not sequentialize completely the execution and leaves scheduling choices.
– The third step completes the *scheduling* of the constrained AEG. Several strategies can be used as long as they are fair. Again, these strategies can be expressed as guarded LTS and scheduling as a synchronized product.
– The fourth step, *concretization*, maps the scheduled (serialized) AEG to a single sequential program. Further transformations (*e.g.,* standard optimizations) can then be carried out on the resulting program.

We have chosen to present fusion in an intuitive and mostly informal way. In particular, we do not provide any correctness proofs. They would require a complete description of the operational semantics of KPN too long to fit space limits. This paper is organized as follows. Section 2 presents the syntax and semantics of KPNs. Section 3 describes AEGs and defines the abstraction and concretization steps which both relate AEG to concrete models (programs and KPNs). Section 4 presents the language of constraints and the two main transformation steps of fusion: constraints enforcement and scheduling. We propose three extensions of the basic technique in Section 5 and, finally, we review related work and conclude in Section 6.

2 Networks

We start by providing the syntax of components and networks. We just outline their semantics and provide some intuition using an example. A complete structural operational semantics for KPNs can be found in [6].

2.1 Basic Components

Components are made of commands c of the form:

$$l_1 : g \mid a \rightsquigarrow l_2$$

where l_1 and l_2 denote labels, g a guard and a an action. An action is either a read operation on an input channel $f?x$, a write operation on an output channel $f!x$, or an internal action i (left unspecified). A component (or process) p is a set of commands $\{c_1, \ldots, c_n\}$. If the current program point of a component p is l_1, if $l_1 : g \mid a \rightsquigarrow l_2$ is a command of p and the guard g is true, then the action a can be executed and the program point becomes l_2.

The components we consider in this paper represent valid, sequential and deterministic programs. They have the following restrictions:

– A component has a unique entry point denoted by the label l_0.
– All the labels used in p are defined in the *lhs* of commands.
– Two commands with the same label have mutually exclusive guards.

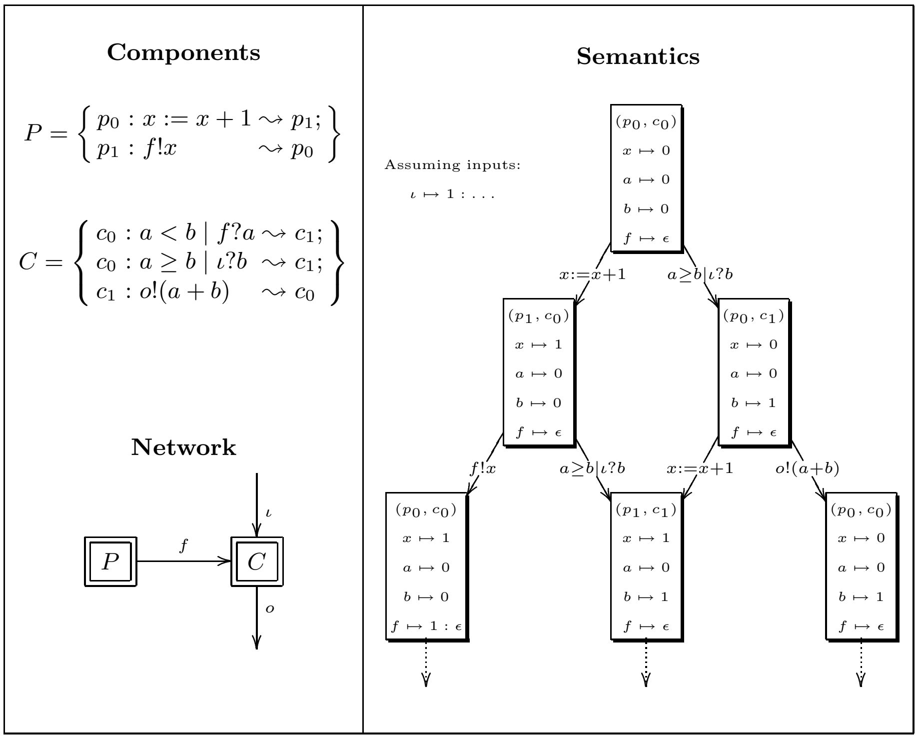


Fig. 2. A Simple Kpn and its trace semantics

The program P in Figure 2 sends the set $\mathbb{N}$ in increasing order on channel f. Program C assigns a with the value read on channel f if $a < b$ or assigns b with the value read on the channel ι otherwise. Then, it sends $a + b$ on the channel o and loops. Note that guards are omitted when they are *true*.

The semantics of a component p is expressed as a Lts $(\Sigma_p, (l_0, s_0), \mathcal{E}_p, \longrightarrow_p)$ where:

- Σ_p is an (infinite) set of states (l, s) where l a label and s a store mapping variables to their values.
- (l_0, s_0) is the initial state made of the initial label l_0 and store s_0. We assume that the initial label is always indexed by 0 and that the initial store initializes integer variables by the value 0,
- $\mathcal{E}_p$ is the set of commands of p,
- $\longrightarrow_p$ is the transition relation (actually, a function) on states labeled with the current command.

The initial labels of programs P and C (Figure 2) are p_0 and c_0 respectively and the variables x, a and b are initialized to 0. In the remaining, we use $c|_g$ and $c|_a$ to denote the guard and the action of the command c respectively. To simplify the presentation, we consider only non-terminating programs. Termination could always be represented by a final looping command such as $l_{end} : skip \rightsquigarrow l_{end}$.

2.2 Networks of Components

A KPN k is made of a set of processes $\{p_1, \ldots, p_n\}$ executed concurrently. Networks are build by connecting output channels to input channels of components. Such channels are called internal channels whereas the remaining (unconnected) channels are the input and output channels of the network. The communication on internal channels is asynchronous (non blocking writes, blocking reads) and is modeled using unbounded fifos. In order to guarantee a deterministic behavior, KPNs require the following conditions [7]:

- An internal channel is written by and read from exactly one process.
- An input channel is read from exactly one component (and written by none).
- An output channel is written by exactly one component (and read from none).
- A component cannot test the absence of values on channels.

In order to simplify technical developments, we assume that networks have a single input and output channels denoted by ι and o respectively and that the input channel never remains empty.

The global execution state of a KPN is called a *configuration*. It is made of the local state of each component and the internal channel states *i.e.,* finite sequences of values $v_1 : \ldots : v_n : \epsilon$.
The operational semantics of KPN is expressed as a LTS $(\Sigma_k, \alpha_0, \mathcal{E}_k, \longrightarrow_k)$ where:

- Σ_k is a (infinite) set of configurations,
- the initial configuration α_0 is such that each component is in its initial state and each internal channel is empty,
- $\mathcal{E}_k$ is the union of the sets of commands of components; these sets are supposed disjoint,
- the transition relation $\longrightarrow_k$ is defined as performing (non deterministically) any enabled command of any process. A command is enabled when the current program point is its *lhs* label, its guard is true in the current configuration/state and it is not a blocking read (*i.e.,* a read on an empty channel).

The transition relation gives rise to an infinite graph representing all the possible execution traces. A small part of the transition relation $\longrightarrow_p$ for our example is depicted in Figure 2. Here, no global deadlock is possible and all traces are infinite.

An infinite execution trace is said to be *fair* if any enabled action at any point in the trace is eventually executed. The denotational semantics of a KPN is given by the function from the input values (the input channel) to the output values (the output channel) generated by fair executions. We will write $Traces(k)$ and $IO(k)$ to denote the set of traces and the denotational semantics of the KPN k respectively. KPNs of deterministic components are deterministic [7]. Also, all fair executions with the same input yield the same output [6]. An important corollary for us is that KPNs are serializable: they can always be implemented sequentially.

3 Abstract Execution Graphs

Network fusion necessitates to find statically a safe and sequential scheduling. This step relies upon an abstract execution graph (AEG), a finite model upper-approximating all the possible executions of the KPN. We present in this section the key properties than an AEG should satisfy and present an example.

An AEG $k^\sharp$ is a *finite* LTS $(\Sigma_{k^\sharp}, \alpha_0^\sharp, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp})$ with:

- $\Sigma_{k^\sharp}$ a finite set of abstract configurations,
- $\alpha_0^\sharp$ is the initial abstract configuration,
- $\mathcal{E}_{k^\sharp}$ a (finite) set of commands,
- $\longrightarrow_{k^\sharp}$ a labeled transition relation.

The idea behind abstraction is to summarize in an abstract configuration a (potentially infinite) set of concrete configurations [10]. This set is given by the function $conc : \Sigma_{k^\sharp} \to \mathcal{P}(\Sigma_k)$ defined as:

$$conc(\alpha^\sharp) = \{\alpha \mid \alpha \approx \alpha^\sharp\}$$

where $\approx$ is a safety relation relating k and $k^\sharp$ (and we write $k \approx k^\sharp$).

There can be many possible abstractions according to their size and accuracy. Network fusion is generic *w.r.t.* abstraction as long as the AEG respect two key properties: *safety* and *faithfulness*. To be safe, the initial abstract configuration of an AEG must safely approximate the initial concrete configuration. Furthermore, if a configuration α_1 is safely approximated by $\alpha_1^\sharp$ and the network evolves in the configuration α_2, then there exists a transition from $\alpha_1^\sharp$ to $\alpha_2^\sharp$ in the AEG such that α_2 is safely approximated by $\alpha_2^\sharp$. These two points ensure that any execution trace of the KPN is safely simulated by one in the AEG. Formally:

Definition 1. **[Safety]** *Let $k \approx k^\sharp$, then $k^\sharp$ is a* safe *approximation of k iff*

$$\alpha_0 \approx \alpha_0^\sharp$$

$$\alpha_1 \approx \alpha_1^\sharp \wedge \alpha_1 \xrightarrow{c}_k \alpha_2 \Rightarrow \exists \alpha_2^\sharp.\ \alpha_2 \approx \alpha_2^\sharp \wedge \alpha_1^\sharp \xrightarrow{c}_{k^\sharp} \alpha_2^\sharp$$

A key property of safe abstractions is that they preserve fairness. Of course, since they are upper approximations they include false paths (abstract traces whose concretization is empty). However, for abstract traces representing feasible concrete traces, fair abstract traces represent fair concrete traces. Safety also implies that all fair concrete traces are represented by fair abstract traces.

An AEG is said to be faithful if each abstract transition corresponds to a concrete transition modulo the non-satisfiability of guards or a blocking read. In other words, faithfulness confines approximations to values. A false path can only be an abstract trace with a transition whose concrete image would be a transition with a false guard or a blocking read. Formally:

Definition 2. [Faithfulness] *Let $k \approx k^\sharp$, then $k^\sharp$ is a* faithful *approximation of k iff*

$$\begin{aligned}\alpha_1^\sharp \xrightarrow{c}_{k^\sharp} \alpha_2^\sharp \ \wedge\ \alpha_1 \approx \alpha_1^\sharp \Rightarrow\ & \exists \alpha_2 . \alpha_2 \approx \alpha_2^\sharp \ \wedge\ \alpha_1 \xrightarrow{c}_k \alpha_2 \\ & \vee\ \neg \mathcal{G}[\![c|_g]\!]\alpha_1 \\ & \vee\ (c|_a = f?x \ \wedge\ \alpha_1[f \mapsto \epsilon])\end{aligned}$$

Faithfulness rules out, for instance, the (highly imprecise but safe) abstraction made of a unique abstract state representing all concrete states. In practice, the same abstract state cannot represent different program points (label configurations).

In order to provide some intuition we give here a crude but straightforward abstraction:

- Each process state is abstracted into the program point it is associated to. So, variables are not taken into account and process stores are completely abstracted away,
- Each internal channel state is represented by an interval approximating the length of its file.

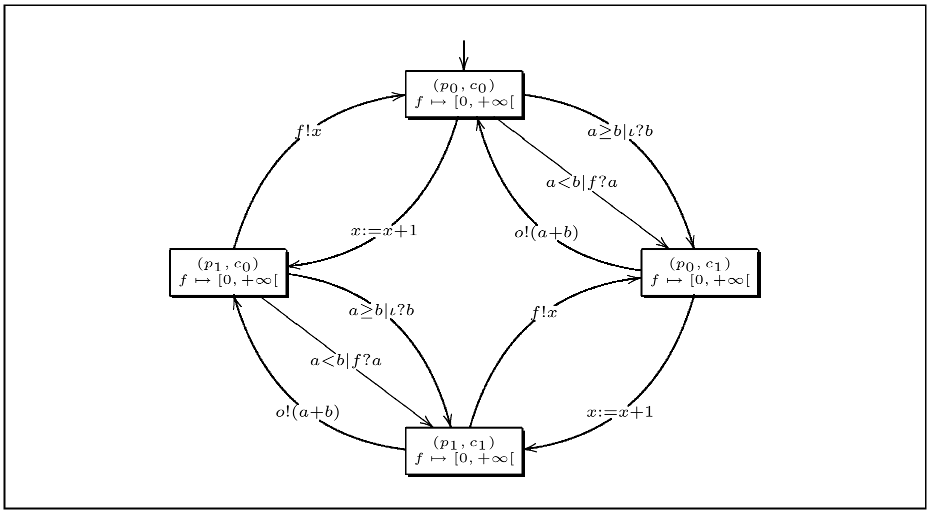


Fig. 3. Example of an AEG

It is the control flow graph of the KPN where each node holds a collection of intervals approximating the lengths of internal channels at the configuration of program points the node represents. The AEG for our running example is given in Figure 3. In this particular example, the state of f is always approximated by the interval $[0, +\infty[$ (the most imprecise information). More precise AEGs could be designed (see Section 5.2).

An AEG bears enough information to be translated back into a program. Commands (guards and actions) label edges and nodes represent labels. The concretization of finite LTS $k^\sharp = (\Sigma_{k^\sharp}, \alpha_0^\sharp, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp})$ into a program is formalized by the following straightforward translation:

$$Concretization(k^\sharp) = \{l_{\alpha_1^\sharp} : c \rightsquigarrow l_{\alpha_2^\sharp} \mid \alpha_1^\sharp \xrightarrow{c}_{k^\sharp} \alpha_2^\sharp\}$$

An important property of safe and faithful abstractions is that their concretization has the same semantics as the network they approximate.

Property 1. If $k^\sharp$ is a safe and faithful approximation of k then $Traces(k) = Traces(Concretization(k^\sharp))$

4 Fusion

The user can specify scheduling constraints defining a subset of execution traces. Constraints impose implementation choices; they serve to guide and to optimize the fusion process. Constraints respect the black box nature of components. They are expressed *w.r.t.* IO operations, liveness properties or sizes of files. When constraints completely sequentialize the execution (no choice remains), they specify a scheduler. In general, however, constraints are incomplete and leave implementation choices.

4.1 Scheduling Constraints

We specify constraints by finite state LTS labeled with guarded actions. Of course, a more user-friendly language for declaring constraints should be studied but this is not the purpose of this article. The formalism used in Sections 2 and 3 is also well-suited to expressing constraints. We enrich the language of guards with two additional constructs dedicated to the expression of scheduling strategies:

$$g_c ::= \overline{f} \ \ominus \ k \mid \mathfrak{B}_p \mid g \quad \text{where } \ominus \text{ is any comparison operator}$$

The size of a channel can be compared against an integer. For instance, $\overline{f} < 5$ is true if the file f has less than 5 elements. The guard $\mathfrak{B}_p$ is true if the process p is blocked (by a read on an empty channel or by other scheduling constraints).

Constraints are more easily specified using sets of actions. We use the following notations:

$$\mathcal{A} ::= \star \mid [f]? \mid [f]! \mid \neg\mathcal{A} \mid \mathcal{A}_1 \cap \mathcal{A}_2 \mid \mathcal{A}_p$$

where

- $\star$ represents any action of the network,
- ? (resp. !) represents any read (resp. write) and f? (resp. f!) any read (resp. write) on file f,
- $\neg\mathcal{A}$ is the complementary set of $\mathcal{A}$,

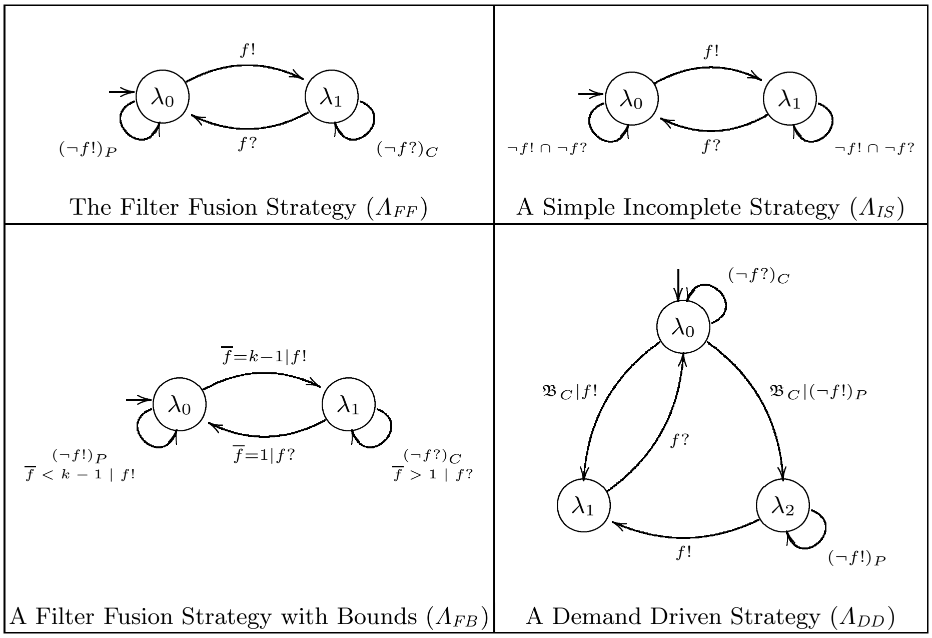


Fig. 4. Examples of Scheduling Constraints

- $\mathcal{A}_1 \cap \mathcal{A}_2$ is the intersection of the sets $\mathcal{A}_1$ and $\mathcal{A}_2$,
- $\mathcal{A}_p$ represents the projection of the set of actions $\mathcal{A}$ onto the commands of the component p.

For instance, $(\neg ?)_p$ represents all non-read actions of component p. Using sets is just more convenient; constraints can be automatically translated into a standard LTS labeled by standard commands afterward.

Figure 4 gathers a few examples of constraints for a network with (at least) two components P (writing a file f) and C (reading the file f).

- The constraint Λ_{FF} summarizes in a small automaton the strategy used by *Filter Fusion* [12]. The producer P starts until it writes on f? The control is passed to the consumer C until it reads f and so on. This strategy bounds the size of the fifo f to be at most 1 and therefore it may introduce artificial deadlocks from some networks. Λ_{FF} sequentializes completely the execution of P and C (no scheduling choice remains).
- The constraint Λ_{IS} is similar to Λ_{FF} except that both P and C can be executed between writes and reads on f. Λ_{IS} leaves some scheduling choices.
- The constraint Λ_{FB} is a generalization of Λ_{FF} to a file f with k places (*i.e.,* P writes k times before the control is passed to C). This is the formalization of the extension of filter fusion proposed in [3]
- A demand driven strategy is specified by Λ_{DD}. The consumer C is executed until it blocks *i.e.,* is about to read the empty channel f. Then, P is executed

until it produces a value in f. The control is passed to C which immediately reads f and continues.

These constraints can be applied to any network as long as it has two components P and C connected at least with a channel f. Of course, constraints can be specified for any number of components and channels.

4.2 Enforcing Constraints

Enforcing a constraint $\Lambda = (\Sigma_\lambda, \lambda_0, \mathcal{E}_\lambda, \longrightarrow_\lambda)$ to an Abstract Execution Graph $k^\sharp = (\Sigma_{k^\sharp}, \alpha^\sharp_0, \mathcal{E}_{k^\sharp}, \longrightarrow_{k^\sharp})$ can be expressed as a parallel composition $(k^\sharp \parallel \Lambda)$. This operation can be defined formally as follows. We assume that all shorthands (like $(\neg ?)_p$) used in constraints are replaced by the actions of the AEG they represent.

$$k^\sharp \parallel \Lambda = (\Sigma_{k^\sharp} \times \Sigma_\lambda, (\alpha^\sharp_0, \lambda_0), \mathcal{E}_{k^\sharp}, \longrightarrow_{k\lambda})$$

with

$$\frac{\alpha^\sharp \xrightarrow{g|a}_{k^\sharp} \alpha^{\sharp\prime} \quad \lambda \xrightarrow{g'|a} \lambda'}{(\alpha^\sharp, \lambda) \xrightarrow{g \wedge g'|a}_{k\lambda} (\alpha^{\sharp\prime}, \lambda')} \qquad \frac{\alpha^\sharp \xrightarrow{c}_{k^\sharp} \alpha^{\sharp\prime} \quad a \in \Sigma_{k^\sharp} \backslash \Sigma_\lambda}{(\alpha^\sharp, \lambda) \xrightarrow{c}_{k\lambda} (\alpha^{\sharp\prime}, \lambda)}$$

If an action a is taken into account by the constraints, the execution can proceed only if both LTS can execute a (*i.e.,* they can both execute commands made of a and a true guard). The actions not taken into account by the constraints can be executed independently whenever possible. Constraints do not introduce new actions ($\mathcal{E}_\lambda \subseteq \mathcal{E}_k$). To simplify the presentation, we assumed in the above inference rules that the guards did not use the condition $\mathfrak{B}_p$. We now present the rule corresponding to this condition in isolation.

The $\mathfrak{B}_p$ construct serves to pass the control to another component when one is blocked. The condition $\mathfrak{B}_p$ is easily defined *w.r.t.* KPNs: p is blocked in configuration α if there is no outgoing transition labeled with a command of p. However, AEGs are approximations with false paths; a component p can be blocked even if the corresponding abstract state has outgoing transitions labeled with commands of p. Actually, p is blocked in an abstract state if any outgoing p transition has either a false guard or is a read on an empty channel (*i.e.,* is not enabled). Formally, let $c_1, \ldots, c_n$ all the commands of p such that $\alpha^\sharp \xrightarrow{c_i}_{k^\sharp} \alpha^\sharp_i$ and $g_i = \begin{cases} \neg(c_i|_g) \ \vee \ \overline{f} = 0 \text{ if } c_i|_a = f?x \\ \neg(c_i|_g) \qquad\qquad \text{otherwise} \end{cases}$
then the necessary and sufficient condition for p to be blocked in $\alpha^\sharp$ is

$$b_p(\alpha^\sharp) = \bigwedge_{i=1,\ldots,n} g_i$$

The product of an AEG with a transition guarded by $\mathfrak{B}_p$ is defined as follows:

$$\frac{\alpha^\sharp \xrightarrow{g|a}_{k^\sharp} \alpha^{\sharp\prime} \quad \lambda \xrightarrow{\mathfrak{B}_p|a}_\lambda \lambda'}{(\alpha^\sharp, \lambda) \xrightarrow{g \wedge b_p(\alpha^\sharp)|a}_{k\lambda} (\alpha^{\sharp\prime}, \lambda')}$$

Figure 5 represents the product of the AEG of Figure 3 with Λ_{FF}. The component P is executed until it produces a value on f then C is executed until it reads a value on f. Note that if $a \geq b$ remains always true then P will never be executed. So, the execution is not fair but it is nevertheless correct and yields the same output as the network (P is never executed only when its production is not needed). The strategy does not use guards, so no new test appears in the constrained AEG. The result is completely sequentialized.

After the constrained AEG is produced, the size of files is reestimated using standard static analysis techniques. We have indicated in Figure 5 the new approximations for $\overline{f}$. They show that the AEG is now bounded (the size of f is at most 1).

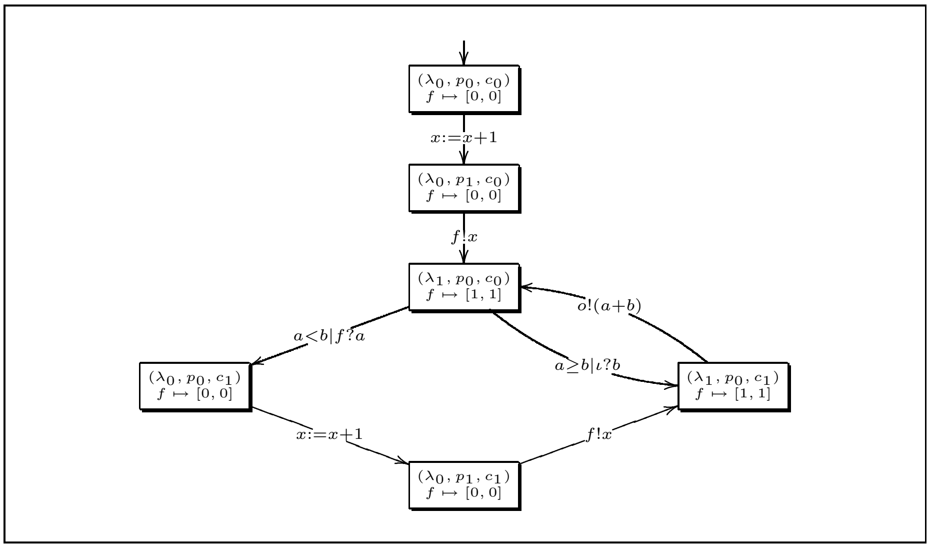


Fig. 5. Fusion with Λ_{FF}

It is easy to check that the AEG can be translated by *Concretization* (see Section. 3) into a sequential program. As already mentioned, one goal of fusion is to suppress internal communications. For unbounded AEG, internal reads and writes are replaced by assignments to lists or fifos. Here, the channel f can be implemented by a single variable v_f and writes $f!x$ and reads $f?a$ by assignments $v_f := x$ and $x := v_f$. These assignments can then be suppressed using standard optimization techniques [1]. Finally, after a renaming of labels, we get:

$$PC = \left\{ \begin{array}{ll} pc_0 : x := x + 1 & \leadsto pc_1; \\ pc_1 : a < b \mid a := x & \leadsto pc_2; \\ pc_1 : a \geq b \mid \iota?b & \leadsto pc_3; \\ pc_2 : x := x + 1 & \leadsto pc_3; \\ pc_3 : o!a + b & \leadsto pc_1; \end{array} \right\}$$

We have presented the parallel composition as a fairly standard automata product. Depending on the size of the LTS, this may cause an unacceptable state explosion. We present a solution to this problem in Section 5.3.

4.3 Scheduling

In general, constraints enforcement leaves implementation choices which must be taken to produce a sequential program. The fusion process makes these choices automatically by scheduling the execution of components. A valid schedule must be fair (all enabled components must be eventually executed) and sequential (the scheduled execution must correspond to a sequential program).

We choose here a simple and fair policy: *round-robin scheduling*. Components are ordered in a circular queue and the scheduler activates them in turn. Either the current active component is blocked (by a read or a user defined constraint) either one of its command is executed. In both cases, the control is passed to the next component. Figure 6 formalizes round-robin for networks with two components P and C as a guarded LTS. It would be easy to generalize such a round-robin LTS for any network with a fixed number of processes.

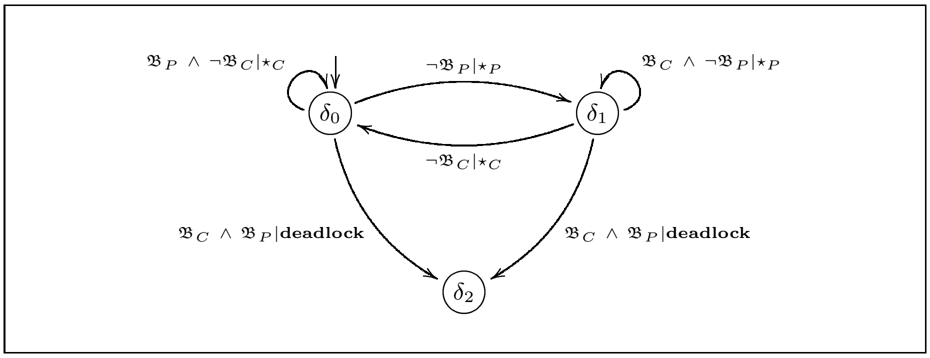


Fig. 6. Round-Robin Scheduling

The schedule is fair and ensures a complete serialization of the execution. It starts by enforcing the execution of one instruction of P, then one instruction of C and so on. If one of the two processes is blocked at its turn, then an instruction of the other process is executed instead. When both processes are blocked then it is a global deadlock denoted by the special instruction **deadlock**.

Constrained AEGs are composed in parallel with the automaton of Figure 6 to obtain sequential programs. The composition is the same as before except that the **deadlock** action does not belong to the set of actions of components. The product will therefore introduce a new **deadlock** transition along with a new state in the AEG. This new transition, which detects a global deadlock, will be implemented by printing an error message and terminating the program.

When such a transition appears in the result of fusion, the user is warned of a possibility of deadlock.

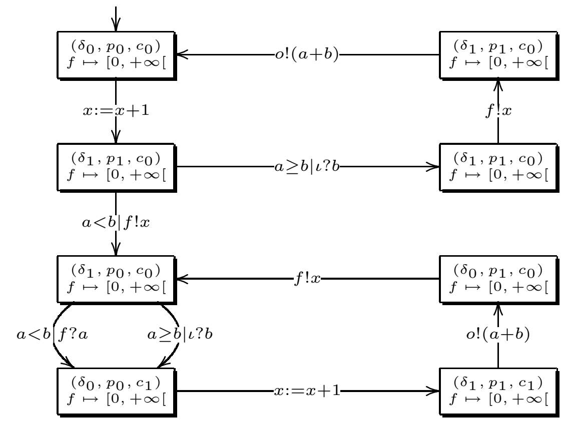


Fig. 7. Sequentialization with Round-Robin

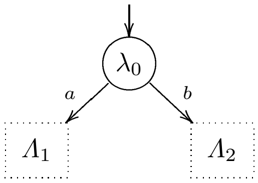
Let us consider the scheduling of the original AEG of Figure 3. This situation would arise if the user does not provide any constraint. The AEG obtained after product (and simplifications) is given in Figure 7. Simplifications are needed since the product of transitions guarded by $\mathfrak{B}_X$ produces many dummy transitions (*i.e.,* with false guards).

The process P is never blocked (it never reads), so the execution can start by $x := x + 1$. The execution must proceed by C if it is not blocked ($\neg\mathfrak{B}_C$). There are two cases: either $a \geq b$ and C is not blocked and its action ($\iota?b$) can be executed, either $a < b$ and C is blocked by a read of the empty file f. In the latter case, round-robin scheduling passes the control to P and executes the action $f!x$. The transition $a < b \mid f!x$ corresponds to "if C is blocked then execute the next P's command". We do not describe any further the product which proceeds similarly. Contrary to the product with Λ_{FF}, the result is fair but unbounded: the data produced by P may accumulate in the channel f without bounds.

The correctness of the scheduling process comes from the fact that the product with Δ_{RR} yields a sequential, fair and faithful AEG. Note that network fusion is generic *w.r.t.* the scheduling strategy. More sophisticated policies (*e.g.,* using several queues, based on static or dynamic priorities, etc.) could be considered as well. As in Section 4.2, we have presented scheduling as an LTS product; scheduling could also be implemented by the technique outlined in Section 5.3.

4.4 Semantic Issues

User-defined constraints can change the semantics of the Kpn. For example, a constraint which bounds communication channels would cause an artificial deadlock into an unbounded Kpn. The user may want to enforce properties even at the price of deadlocks. We consider that a change of semantics is acceptable as long as it depends on the user and remains under her or his control. However, this requires to restrict the class of acceptable constraints. Consider the following constraint:



where a and b are two non-mutually exclusive commands and Λ_1 and Λ_2 represents distinct constraints. Since a and b can be executed indifferently, a choice will be made by scheduling. However, depending on this choice, the constraints that are enforced afterwards (Λ_1 or Λ_2) are different. For example, Λ_1 may imply an artificial deadlock in some (non statically determined) cases and Λ_2 in some other cases. In other words, the semantics of the resulting program will depend on a blind choice made by fusion. This semantic change is not acceptable since it would be out of the control of the user.

Our solution is to restrict the class of acceptable constraints. Namely, if a constraint leaves a non deterministic choice such as a or b above then the constraints must ensure that all processes can still evolve in the same way (an artificial deadlock in one side implies that we have the same artificial deadlock on the other side). Each choice a or b correspond to set (language) of acceptable traces ($a.\mathcal{L}(\Lambda_1)$ and $b.\mathcal{L}(\Lambda_2)$). A constraint is acceptable if for each choice, the projection of the corresponding languages to the commands of any process are equivalent. For the above example we must enforce that

$$\forall p.\ \ a.\mathcal{L}(\Lambda_1) \downarrow p = b.\mathcal{L}(\Lambda_2) \downarrow p$$

With this condition, the choices made by the scheduling step do not have any semantic impact.

All the constraints of Figure 4 are acceptable. It is obvious for Λ_{FF}, Λ_{FB}, Λ_{DD} since they do not leave any non deterministic choice. In Λ_{IS}, the two transitions labeled by $\neg f! \cap \neg f?$ leave the choice between executing P or C. However, in both states, they lead to the same state (and therefore accept the same language).

5 Extensions

The preceding sections have presented the main ideas of network fusion. We hint here at three ways of extending the basic technique: providing more linguistic support to the user, working on more precise abstractions, avoiding products between LTS. These three extensions all aim at getting more efficient fused programs.

5.1 Linguistic support

Scheduling constraints allow users to control network fusion. Other linguistic support could be provided to users as well. We focus here on special commands allowing to alleviate the *false path problem*. False paths arise when data depending controls are abstracted by non deterministic choices [2]. This standard approximation makes fusion consider infeasible paths and spurious deadlocks.

Synchronization using data values	**Synchronisation using linguistic extensions**
$P = \left\{\begin{array}{ll} l_0 : \iota?N & \rightsquigarrow l_1; \\ l_1 : ct!N & \rightsquigarrow l_2; \\ l_2 : i := 0 & \rightsquigarrow l_3; \\ l_3 : i < N \mid i = i+1 & \rightsquigarrow l_4; \\ l_3 : i \geq N \mid skip & \rightsquigarrow l_0; \\ l_4 : dt!x & \rightsquigarrow l_3 \end{array}\right\}$	$P = \left\{\begin{array}{ll} l_0 : \iota?N & \rightsquigarrow l_1; \\ l_1 : ct!N & \rightsquigarrow l_2; \\ l_2 : i := 0 & \rightsquigarrow l_3; \\ l_3 : i < N \mid i = i+1 & \rightsquigarrow l_4; \\ \mathbf{l_3 : i \geq N \mid wait(dt)} & \rightsquigarrow \mathbf{l_0}; \\ l_4 : dt!x & \rightsquigarrow l_3 \end{array}\right\}$
$C = \left\{\begin{array}{ll} l_0 : o!0 & \rightsquigarrow l_1; \\ l_1 : ct?M & \rightsquigarrow l_2; \\ l_2 : j := 0 & \rightsquigarrow l_3; \\ l_3 : j < M \mid j = j+1 & \rightsquigarrow l_4; \\ l_3 : j \geq M \mid skip & \rightsquigarrow l_0; \\ l_4 : dt?y & \rightsquigarrow l_5; \\ l_5 : o!y & \rightsquigarrow l_3 \end{array}\right\}$	$C = \left\{\begin{array}{ll} l_0 : o!0 & \rightsquigarrow l_1; \\ \mathbf{l_1 : \neg waiting?(dt) \mid dt?y} & \rightsquigarrow \mathbf{l_3}; \\ \mathbf{l_1 : waiting?(dt) \mid proceed(dt)} & \rightsquigarrow \mathbf{l_0}; \\ l_2 : o!y & \rightsquigarrow l_1 \end{array}\right\}$

Fig. 8. A false path problem (left) and its solution (right)

The left part of figure 8 shows a simple but characteristic example of the problem. The process P begins by sending on channel ct the number of items it will produce on channel dt. Then, P and C respectively writes and reads the same number of items on dt ($M = N$). However, this information is lost in the AEG which abstracts away values. The fusion process must therefore consider the case where P produces not enough values and C is blocked and also the case where P produces unconsumed values and the size of dt cannot be bounded.

This problem can be alleviated using commands making synchronization or termination explicit. The languages of actions and guards are extended with the following constructs:

$$\begin{array}{l} a ::= \mathbf{wait}(f) \mid \mathbf{proceed}(f) \mid \ldots \\ g ::= \mathbf{waiting}?(f) \mid \ldots \end{array}$$

- The commands **wait**(f) and **proceed**(f) permit to express a rendezvous between the producer and the consumer of a file f. The producer blocks on **wait**(f) until the consumer emits **proceed**(f).
- The predicate **waiting**?(f) evaluates to true if the producer is waiting on **wait**(f) and all data has been consumed on f. It evaluates to false if there is some available data. It blocks if there is no available data and the producer is not waiting.

These instructions are just syntactic sugar and **waiting**?(f) do not affect the determinism of KPNs. They could be implemented by writing/reading a special value on an additional channel. On the other hand, they do provide more information to the fusion process and permit to avoid false paths.

The right part of Figure 8 shows how to take profit of these instructions on the previous example. Instead of communicating via ct the number of items written on dt, P finishes its emission by waiting to C. The consumer reads until it has consumed all data produced by P; it then releases P and both processes may proceed. Such explicit rendezvous can be taken into account by the abstraction step to avoid the problematic false paths mentioned above.

Others instructions could be considered as well. For example, an instruction **close**(f), indicating that a process will not write or read on f anymore, would also be useful.

5.2 More precise abstractions

In section 2, we presented an abstraction representing the control flow graph of the KPN. This abstraction gives a very imprecise approximation for the size of file f at each state ($[0;+\infty[$). As long as they respect the safety and faithfulness properties, many other abstractions could be used. We present here a new abstraction aimed at finding bounded schedules when they exist. A bounded schedule ensures that the size of fifo files remain bounded throughout the execution. In this case, fifo can implemented (after fusion) by local variables (instead of dynamically allocated data structures). Furthermore, when the precise size of a channel is known there is no need to test for its emptiness before reading it.

In some contexts, such as embedded systems, it is crucial to find bounded schedules and lot of work has been devoted to this issue ([11], [5], [13]). In the context of Petri Nets, Cortadella *et al.* presented a new criterion which limits the search state for schedules [5] . They conjectured that if a bounded schedule exists then it will be found in the delimited search space. We can adapt the criterion to our context to produce abstractions suited to the discovery of bounded schedule.

The idea is to have precise abstract states associating an integer to each size (not an interval anymore). The same set of control points (*e.g.,* (p_0, c_1)) may appear several times in the AEG with different sizes of files. The finiteness of the AEG is ensured by the *irrelevance* criterion. A state s_2 is said irrelevant if the AEG contains another state s_1 such that:

- s_2 is reachable from s_1,

– all fifo have their size in s_2 greater or equal than their size in s_1,
– each fifo whose size is greater in s_2 than in s_1 has a non-zero size in s_1.

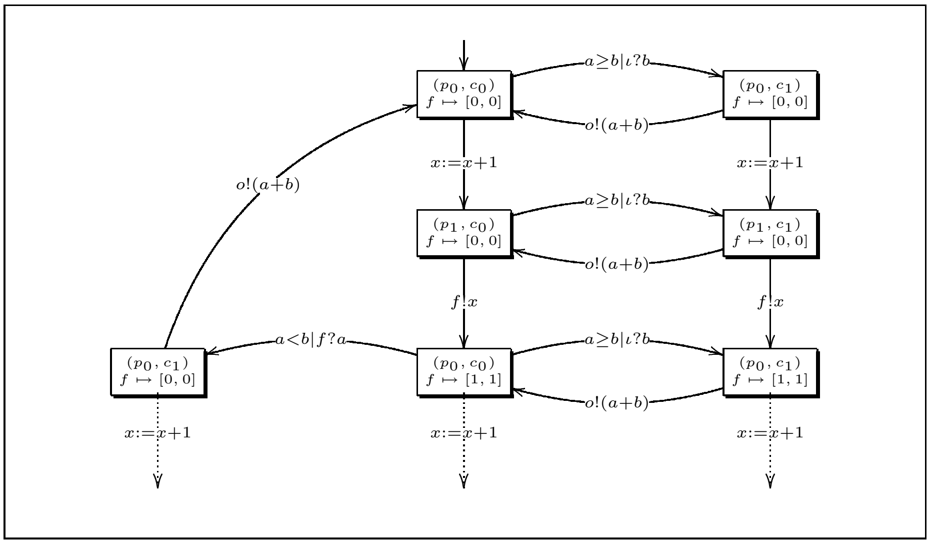


Fig. 9. AEG with irrelevance criterion (excerpt)

The idea behind this criterion is that a irrelevant state cannot enable any new action (*e.g.,* it does not enable a blocked read). It is no use to continue unfolding the graph. It can be closed using a state where the size of channels are approximated by $[0; +\infty[$. A small part of the AEG obtained using the irrelevance criterion on our running example is shown on Figure 9. A bounded schedule can be found in the part shown. This improved precision allows to find bounded schedules automatically; it also involves larger AEGs.

5.3 Instrumented Product

Two steps of network fusion are described as a synchronized product between guarded LTS. Obviously, in some cases, this could cause a state explosion and produce too large programs. A solution to avoid this space problem is to implement the product by instrumenting the AEG. The LTS representing the constraints or the schedule is taken into account by the AEG by introducing a variable (to represent the state of the LTS) and new instructions (to represent state transitions).

We have used such a technique in [4] to enforce safety properties (expressed as finite state automata) on programs. We have shown that the instrumentation can be made very efficient using simple techniques (specialization, minimization and reachability analysis). This instrumented product introduces at worst an assignment (a state transition) at each *if* and *while* command.

This technique is easily extended to guarded LTS. Figure 10 represents the result of the instrumented product between the AEG of Figure 3 and Λ_{FF}. It has the same number of states as the original AEG. On the other hand, instructions ($l := \{0, 1\}$) and tests ($l = \{0, 1\}$) have been inserted to encode the state transitions of Λ_{FF}. Compared to the LTS of Figure 5 which represents the standard synchronized product, some states like (λ_0, p_0, c_0) and (λ_1, p_0, c_0) are now merged into a single state (p_0, c_0). Transitions from this state must now test whether underlying LTS Λ_{FF} is in the state λ_0 or λ_1.

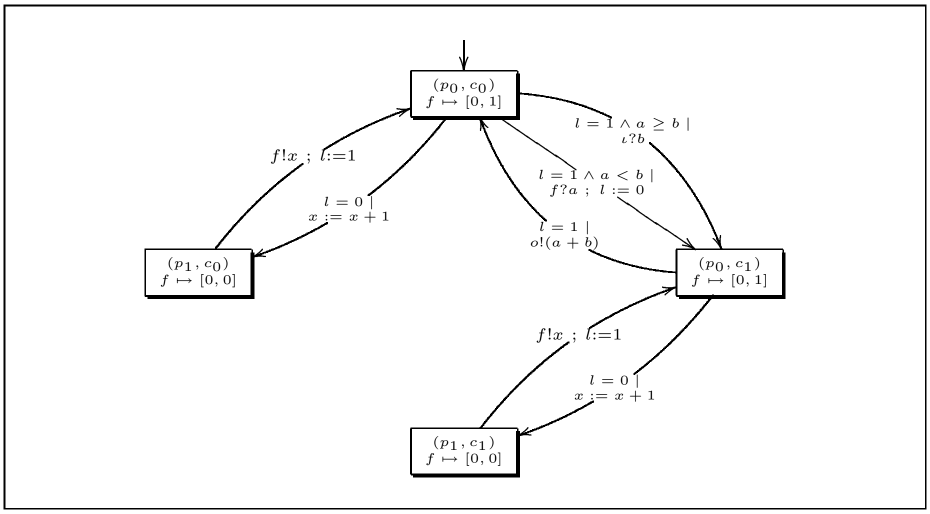


Fig. 10. Instrumented product with Λ_{FF}

On this example, the smaller number of states is certainly not worth the overhead. In general, however, instrumented product is at most linear in size whereas synchronized product may entail a quadratic blowup. A small time overhead is preferable to a space explosion. In any case, the user should be given the opportunity to specify on which LTS (or on which parts of a LTS) using standard or instrumented product.

6 Conclusion

The inspiration and motivation for this work came from two main sources:

- Filter fusion [12], a simple algorithm to merge a producer connected by a single channel to a consumer. Filter fusion is restricted to very specific networks (pipelines) and to a fixed strategy. Our work can be seen as a formalization of filter fusion using synchronized product and as well as a generalization to arbitrary networks and user-defined strategies. The application of our technique on pipelined filters with the constraint Λ_{FF} (Figure 4) is equivalent to filter fusion. The extension of filter fusion with a more sophisticated scheduling strategy proposed in [3] is formalized in our framework by the scheduling constraints Λ_{FB} (Figure 4).

– Aspect-Oriented Programming (AOP) [8] whose goal is to isolate aspects (*e.g.,* security, synchronization, etc.) that cross-cut the basic functionality of the program. Our scheduling constraints can be seen as a scheduling/synchronization aspect and their enforcement as aspect weaving. As in [4], we consider aspects specified as temporal formulas on the trace semantics of programs. In network fusion, aspects express scheduling and synchronization choices by filtering unwanted (or selecting desired) execution traces. This restricted and formal view permits to describe and control precisely the semantic impact of weaving (usually, a very difficult task in AOP).

Some functional program transformations bear similarities with network fusion. As fusion aims at removing values produced on channels by the composition of components, Listlessness [15] and deforestation [16] aim at removing the intermediate data structures produced by the composition of functions. As filter fusion, these transformations consider producer-consumer pairs and have a fixed fusion strategy.

The area of embedded/reactive systems has produced a large body of work on static scheduling. Lin [9] studies the static scheduling of synchronously communicating processes. Cortadella *et al.* [5] and Strehl *et al.* [13] consider scheduling of asynchronous process networks. They all use petri nets as their underlying formalism. Like Parks [11], they focus on bounded scheduling and do not consider user-defined constraints even if some integrate a form of fusion. Strehl *et al.* [14] propose a design model that permits the specification of components and scheduling constraints. They derive a scheduler but do not consider fusion.

We have presented a generic and flexible framework for merging networks of interacting components. It is based on guarded labeled transition systems, synchronized product, static analysis and transformation techniques. Fusion can be applied to a large class of networks (KPNs) and can take into account user-defined scheduling constraints. The technique can be parameterized by different abstractions, constraints and scheduling strategies. Still, a lot of work remains to be done.

The formalization and the correctness proofs should be completed. A prototype needs to be implemented in order to validate the approach experimentally. We expect that large programs can be abstracted into small automata since fusion focuses on I/O instructions (blocks of internal instructions can be represented by a single action). Along with the use of instrumented product in problematic cases, we are confident that efficient and reasonable sized programs can be produced.

More generally we see network fusion as part of a more general framework to assemble and fuse components. A first feature of such a framework would consist in an architecture description language to specify the assembly (*i.e.,* the ports and their connections). Another useful feature would be the ability to specify the synchronization instructions. They do not have to be IO instructions as supposed previously. By considering some actions of two components as IO operations on a (conceptual) channel f (*i.e.,* $f!x$ and $f?x$), it becomes possible to impose constraints on their interleaving.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. Arrigoni, Duchini, and Lavagno. False path elimination in quasi-static scheduling. In *Automation and Test in Europe Conference and Exhibition (DATE'02)*, pages 964–970, 2002.
3. R. Clayton and K. Calvert. Augmenting the proebsting-watterson filter fusion algorithm, 1997. http://citeseer.ist.psu.edu/186288.html.
4. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 54–66, 2000.
5. J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. L. Sangiovanni-Vincentelli. Task generation and compile-time scheduling for mixed data-control embedded software. Technical Report LSI-99-47-R, Dept. of Software, Universitat Politecnica de Catalunya, 1999.
6. M. Geilen and T. Basten. Requirements on the execution of Kahn process networks. In *European Symposium on Programming (ESOP'03)*, 2003.
7. G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress (Information Processing'74)*, pages 471–475, 1974.
8. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the European Conference on Object-Oriented Programming*, June 1997.
9. B. Lin. Software synthesis of process-based concurrent programs. In *Proceedings of the 1998 Conference on Design Automation (DAC-98)*, pages 502–505, 1998.
10. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
11. T. M. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California, Berkeley, 1995.
12. T. A. Proebsting and S. A. Watterson. Filter fusion. In *Symposium on Principles of Programming Languages (POPL'96)*, pages 119–130, 1996.
13. K. Strehl, L. Thiele, D. Ziegenbein, and R. Ernst. Scheduling hardware/software systems using symbolic techniques. In *Proceedings of the seventh international workshop on Hardware/software codesign (CODES '99)*, pages 173–177, 1999.
14. L. Thiele, K. Strehl, D. Ziegenbein, R. Ernst, and J. Teich. Funstate - an internal design representation for codesign. In *International Conference on Computer-Aided Design (ICCAD '99)*, pages 558–565, 1999.
15. P. Wadler. Listlessness is better than laziness. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1984.
16. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.