Noname manuscript No.
(will be inserted by the editor)

Modular Static Scheduling of Synchronous Data-flow Networks

An efficient symbolic representation

Marc Pouzet · Pascal Raymond

Received: January 2010 / Accepted: date

Abstract This paper addresses the question of producing modular sequential imperative code from synchronous data-flow networks. Precisely, given a system with several input and output flows, how to decompose it into a minimal number of classes executed atomically and statically scheduled without restricting possible feedback loops between input and output?

Though this question has been identified by Raymond in the early years of Lustre, it has almost been left aside until the recent work of Lublinerman, Szegedy and Tripakis. The problem is proven to be intractable, in the sense that it belongs to the family of optimization problems where the corresponding decision problem — there exists a solution with size c — is NP-complete. Then, the authors derive an iterative algorithm looking for solutions for $c = 1, 2, ...$ where each step is encoded as a satisfiability (SAT) problem.

Despite the apparent intractability of the problem, our experience is that real programs do not exhibit such a complexity. Based on earlier work by Raymond, the current paper presents a new encoding of the problem in terms of input/output relations. This encoding *simplifies* the problem, in the sense that it rejects some solutions, while keeping all the optimal ones. It allows, in polynomial time, (1) to identify nodes for which several schedules are feasible and thus are possible sources of combinatorial explosion; (2) to obtain solutions which in some cases are already optimal; (3) otherwise, to get a non trivial lower bound for c to start an iterative combinatorial search. The method has been validated on several industrial examples.

Revised and extended version of [14]. This work is supported by the SYNCHRONICS large scale initiative of INRIA.

Marc Pouzet
École Normale Supérieure and Université Pierre et Marie Curie.
Physical address: École Normale Supérieure, 45 rue d'Ulm, 75230 Paris cedex 05, France.
E-mail: Marc.Pouzet@ens.fr.

Pascal Raymond
Laboratoire VERIMAG, 2 avenue de Vignate, 38610 Gières, France.
E-mail: Pascal.Raymond@imag.fr.

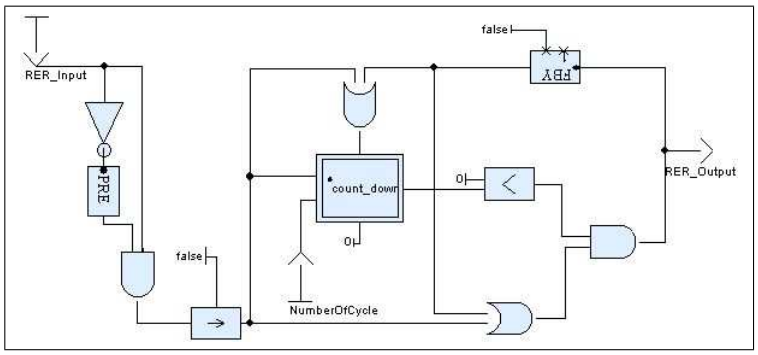


Fig. 1 A SCADE (v5) block-diagram

The solution applies to a large class of block-diagram formalisms based on atomic computations and a *delay* operator, ranging from synchronous languages such as LUSTRE or SCADE to modeling tools such as SIMULINK.

Keywords Real-time systems, Synchronous languages, Block-diagrams, Compilation, NP-completeness, Partial orders, Preorders

1 Introduction

The synchronous block-diagram or *data-flow* formalism is now preeminent in a variety of design tools for embedded systems. Sequential code generation of synchronous block-diagrams have been considered in the early years of LUSTRE [7] and SIGNAL [1] and is provided by industrial tools such as SCADE [1] and RTBUILDER [2] for almost fifteen years. Though it has been considered more recently, modeling and simulation tools such as SIMULINK [3] and MODELICA [4] are now equipped with automatic code generators.

We focus here on the problem of generating imperative, sequential code, implementing the functional behavior of a parallel data-flow network. We keep abstracted the (somehow orthogonal) problem of data management, that is, how values are actually passed from one node to another and even the interpretation of each operator. In particular, we address both data-flow networks with discrete-time (e.g., SCADE) or continuous-time (e.g., SIMULINK) semantics. Figure 1 gives an example of a SCADE block-diagram and Figure 2, an example of a SIMULINK one.

Whatever be the semantics of nodes in a network, there are basically two types of atomic nodes. *Instantaneous* nodes need to evaluate all their arguments in order to produce their outputs (e.g., combinatorial functions). On the contrary, *delay* nodes are able to produce their outputs before reading their inputs. They correspond to unitary registers in synchronous designs (the so-called `pre` operator of LUSTRE), initialized buffers in Kahn process networks [9,10] or continuous integrators in SIMULINK. Delays

[1] `http://www.esterel-technologies.com/scade/`
[2] `http://www.geensoft.com/en/article/rtbuilder`
[3] `http://www.mathworks.com/product/simulink`
[4] `http://www.modelica.org`

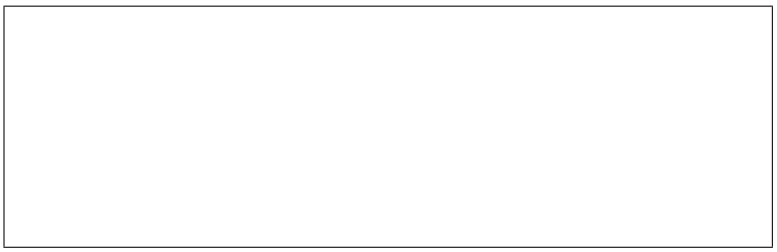


Fig. 2 A SIMULINK block-diagram

are essential to cut instantaneous data-dependencies and to restrict to well-founded feedback loops.

Given a system described as a data-flow network of atomic operators, code generation aims at producing a static schedule satisfying data-dependencies. This static schedule is feasible when there is no combinatorial loop, i.e., every loop in the graph crosses a delay. Data-flow formalism, just like any advanced programming language, allow the user to abstract his own programs into reusable components, that is, to build modular functions. This raises the problem of defining what is exactly a reusable parallel program, and how to compile it once and for all into sequential code. As for sequential languages, modular (or separate) compilation produces a single sequential procedure, executed atomically, for every block-diagram definition. Nonetheless, this modular code generation is not always feasible even in the absence of combinatorial loops, as noticed by Gonthier [6]: if *copy* is a node defined by $copy(x, y) = (x+1, y+1)$, then the equation $(y, z) = copy(t, y)$ defines two valid streams y and z (since $y = t + 1$ and $z = y+1 = (t+1)+1$) but it cannot be statically scheduled if *copy* is compiled into one atomic step function. Indeed, it would make both y and z depend on t and y. This observation has led to two main compilation approaches. With the first one, or *white-boxing*, nodes are statically inlined before code generation. This is the solution taken in the academic LUSTRE compiler. The opposite solution or *black-boxing*, keeps maximal code sharing by compiling each node individually into *one* step function executed atomically. From the user point of view, a node is considered as instantaneous whichever are the actual dependencies between its inputs and outputs. As a consequence, every feedback loop must cross an explicit delay outside of the node. This compilation method rejects causally correct programs (such as the *copy* example) which would have worked properly in a parallel execution or compiled with a white-box technique. This solution is the one taken in the industrial compiler of SCADE and is combined to inlining, on demand [5], to accept all causally correct programs [3].

In this paper, we investigate an intermediate approach between the two former ones, which we call *grey-boxing*. It is based on the observation that some nodes are *compatible* in the sense that they can be executed together without restricting possible feedback loops between inputs and outputs. Then, the data-flow graph can be partitioned into a minimal number of classes, each of them executed atomically. The idea of grey-boxing originally appeared in a work by Raymond published in a French report in 1988 [15]. Left aside, the subject has been reconsidered recently by Lublinerman and Tripakis. Their first proposal [13] was based on dynamic testing to avoid

[5] This is manually controlled by the designer with a compilation flag.

the re-computation of shared values but, as they noticed in [12], this solution is costly in term of code size and execution time. In this recent work, Lublinerman, Szegedy and Tripakis come back to the original optimal static scheduling problem. They prove that the problem is intractable since it belongs to the family of optimization problems where the corresponding decision problem — is there a solution with at most c classes? — is NP-complete as it encodes the *clique cover* in a graph. Then, the authors derive an iterative algorithm looking for solutions for $c = 1, 2, ...$ where each step is encoded into a SAT problem.

Despite the apparent intractability of the problem, our experience is that real programs do not exhibit such a complexity. This calls for a careful look at the sources of combinatorial explosion and for an efficient encoding which limits this possible explosion. Moreover, we would like to build a polynomial test checking whether a given instance of the problem can be polynomially solved or fall into the general case, and thus, requires an enumerative search. Based on earlier work by Raymond on a study of input/output relations, this paper proposes a symbolic and efficient representation of the problem. We show that this encoding *simplifies* the problem in the sense that it rejects solutions but keeps *all* the optimal ones. This symbolic representation allows, in polynomial time, (1) to identify nodes for which several schedules are feasible and thus are possible sources of complexity; (2) to obtain solutions which in some cases are already optimal; (3) otherwise, to get a non trivial lower bound for c to start an iterative combinatorial search, once the symbolic representation is translated into a boolean formula. All this is proven using basic properties of partial orders.

The paper is organized as follows. Section 2 gives an overview of the optimal scheduling problem and Section 3 develops the formalization. We start by giving another proof that optimal scheduling is intractable to help clarifying the sources of combinatorial explosion. In section 4, we present the symbolic representation based on input/output analysis, related properties and algorithms. Section 5 gives experimental results. We discuss related works in Section 6 and we conclude in section 7. The source code of the OCaml implementation is given in appendix A.

2 Overview

2.1 From Synchronous Data-flow to Relations

Consider the following synchronous data-flow program given below in Lustre syntax. The corresponding block-diagram is given on the left side of Figure 3. The block `NODE` has two inputs `a`, `b` and two outputs `x` and `y`, all with data-type `ty`. The body is made of two equations defining respectively `x` and `y`.

```
node NODE(a, b: ty) returns (x, y: ty);
  let
    x = j(a, f(D(a), b));
    y = h(b);
  tel
```

There are essentially two kinds of operators in such a network, instantaneous and delay operators. Instantaneous operators (`j`, `f` and `h`) need their current inputs in order to produce their current output whereas a delay (`D`) is able to produce its output *before*

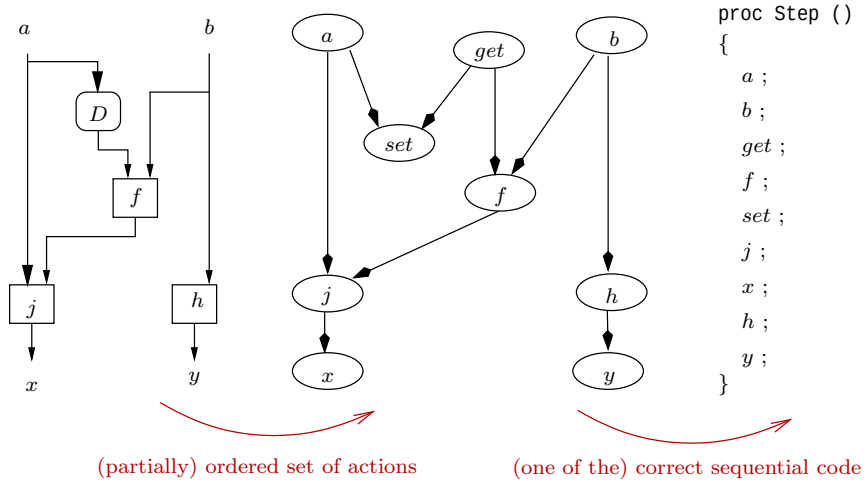


Fig. 3 A data-flow network and the corresponding ordered set of actions

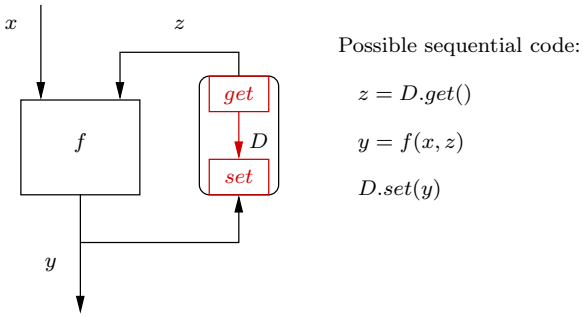


Fig. 4 The get/set principle breaks feedback loops

it reads its input [6]. Delay nodes are used to program dynamical systems with feedback loops.

Focusing only on the scheduling problem of a data-flow network with delays — and not memory representation and optimization — the simplest way to represent a delay D is to express it as two atomic imperative actions $D.get$ and $D.set$. $D.get$ returns the current value of the delay whereas $D.set$ reads the current input and stores it for remaining execution. More importantly, if the result of $D.get$ is necessary, it must be executed before $D.set$ is executed. In other words, a delay *reverses* the data-flow dependency between its input and its output and thus breaks cycles (Figure 4). Using this lower-level representation, we consider data-flow networks that are simply partially ordered sets of atomic actions as defined below.

[6] The unitary delay is concretely written `pre` in Lustre and $1/z$ in (discrete) Simulink. From the dependencies point-of-view, an integrator $(1/s)$ for a continuous signal in Simulink acts similarly.

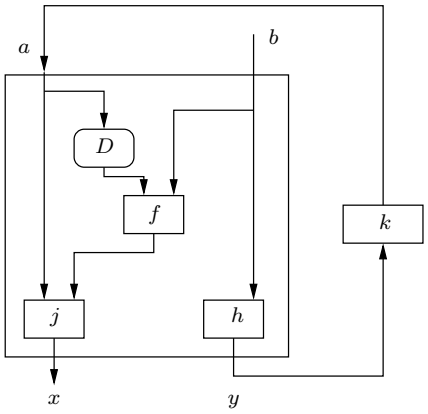


Fig. 5 A correct feedback use of a parallel component

Definition 1 (Abstract Data-flow Networks) A system $(A, I, O, \preceq)$ is made of:

1. a finite set of actions A,
2. a subset of inputs $I \subseteq A$,
3. a subset of output $O \subseteq A$ (not necessarily disjoint from I)
4. and a partial order $\preceq$ to represent the precedence relation between actions.

There is no constraint on I and O with respect to $\preceq$. In particular, one may have $x \preceq i$ or $o \preceq x$ with $x \in A$, $i \in I$ and $o \in O$.

In the sequel, the static scheduling problem of a concrete data-flow program is considered only on this representation. We shall not describe further the transformation encoding from a high level language (like SCADE or SIMULINK) to the abstract model $(A, I, O, \preceq)$.

Figure 3 shows a block-diagram (left) and the corresponding ordered set of actions (center). The delay node D is replaced by the corresponding actions *set* and *get*. Note that only the direct dependencies are shown, the partial order is, as usual, the transitive closure of the represented acyclic graph. From this partially ordered set, code generation mainly consists in finding a correct schedule for the actions, that is, a total order including the partial dependency order. One correct schedule is shown on the right-hand side of Figure 3.

2.2 Feedback Loops and Grey-boxing

In such a network, it makes sense to feed an output back to an input as soon as the output does not depend combinatorially on this input. We say that such a feedback is causality correct. This is illustrated by the example of Figure 5. If we allow such an external loop, it is not possible to use a single, monolithic code for `NODE`, such as the one given on the right of Figure 3. In any correct schedule, the computation of the external node k must take place between the computations of the internal nodes h and j. As a consequence, at least two atomic pieces of sequential code (or sequential blocks) are necessary for executing a step in any context: one for h and one for j.

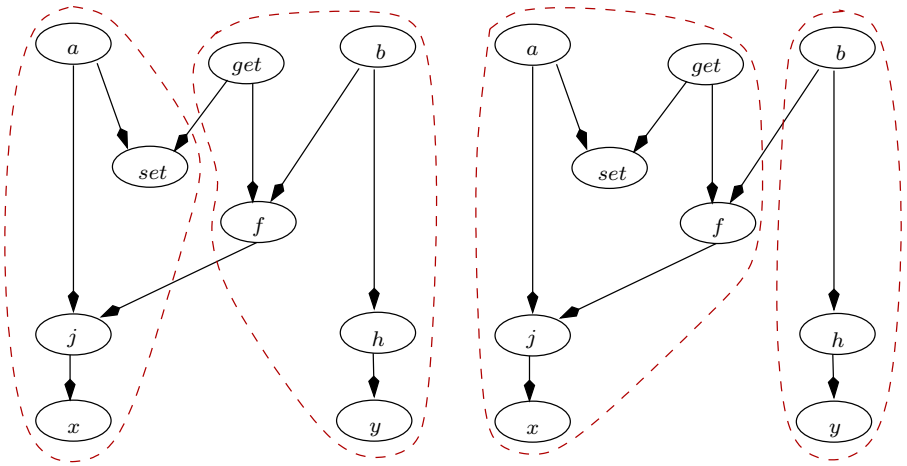


Fig. 6 Two possible "grey-boxing" and optimal scheduling

However, it is not necessary to introduce a block for each internal variable, which would be equivalent to inlining (or white-boxing). As a matter of fact, some internal nodes can be gathered with either h or j without preventing causally correct feedback loops. For example, whatever the calling context, f can be computed:

- together with h, as soon as b is provided;
- together with (and before) j in order to provide the output x.

The same reasoning holds for the delay D:

- $D.get$ is clearly associated to the computation of f;
- $D.set$ requires the input a, and for this reason, is related to the action j.

Finally, this intuitive reasoning shows that the node can be abstracted into two blocks of code:

- the "class" of h with input b and output y;
- the "class" of j with input a, output x, and which must always be computed after the class for `h`.

The frontier between these two classes is not strict: some internal computations can be performed on one side or the other, like g and f. Figure 6 shows two particular solutions, each of them with two classes. Note that this number of classes is *optimal*, since it is impossible to produce only one class without forbidding the correct feed-back from y to a.

The problem of partitioning a data-flow program into a (minimal) number of sequential blocks is called the *(Optimal) Static Scheduling Problem*.

Figure 7 summarizes this principle on our example:

- an optimal partition of the actions is chosen (left),
- a user interface (center) is derived from the partition, which contains:
  - the scheduling information: a step of `NODE` is performed by two atomic actions $N1$ and $N2$. $N1$ must be performed before $N2$.
  - the input/output "pins" recalling how the program can be connected by a caller.
- some total order is chosen for each block, giving the actual sequential code (right).

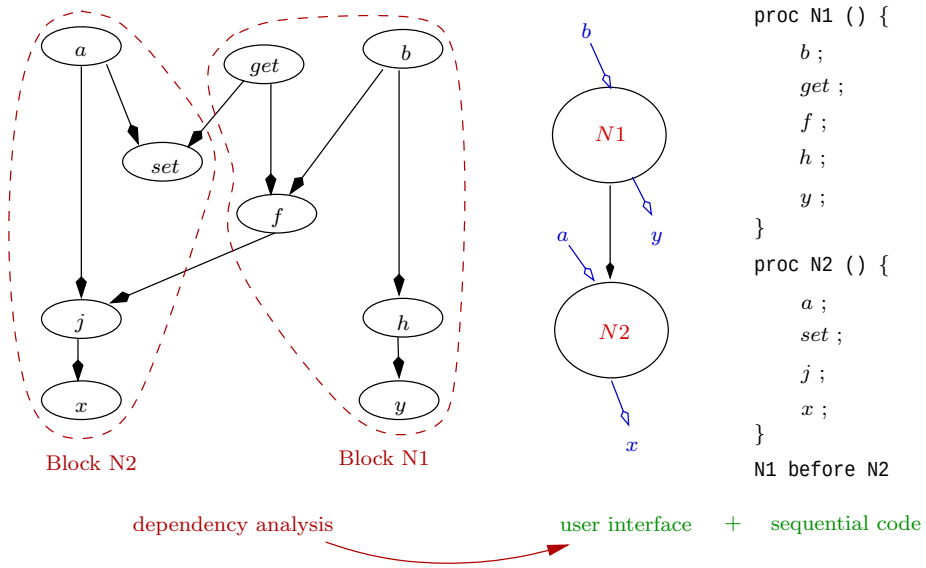


Fig. 7 Grey-boxing: orderer blocks plus sequential code

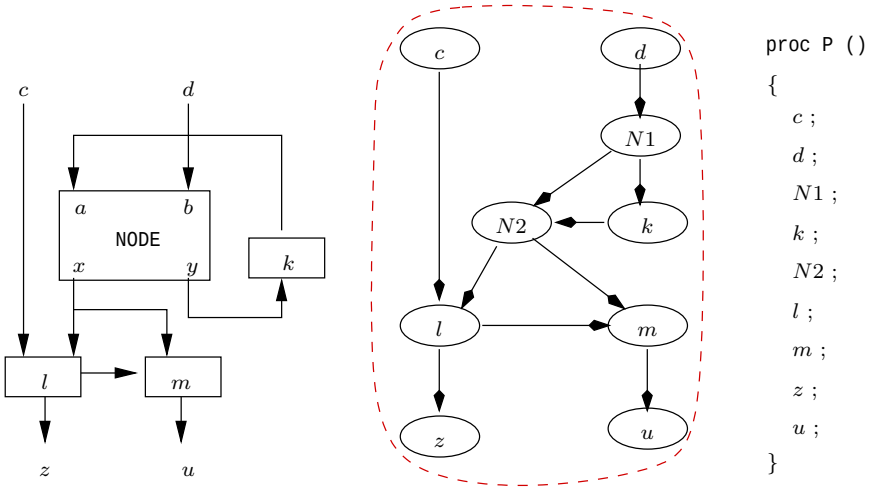


Fig. 8 Modular static scheduling

2.3 Modularity

The formulation with $(A, I, O, \preceq)$ is fully modular and sufficient to address the static scheduling problem.

At each level of the hierarchy, we can focus on a set of actions, some of them being inputs and/or outputs and some being local. Consider the example of Figure 8:

the block `NODE` is used within a block (P), with a feedback loop similar to the one of Figure 5. According to the informations computed during the compilation of `NODE` (Figure 7), the block-diagram (left) is interpreted as a partially ordered set of actions (center). The grey-boxing principle is applied, and as it appears that all outputs are depending on all inputs, no external feedback is possible without creating a combinational loop. As a consequence, the node P can be implemented by a single block. A correct sequential code is produced (right).

3 Formalization

3.1 Optimal Static Scheduling

Let A be a set of actions, partially ordered by $\preceq$, and two subsets $I \subseteq A$ (input nodes) and $O \subseteq A$ (output nodes).

Definition 2 (Compatibility) Two actions $x, y \in A$ are said to be compatible regarding static scheduling (noted $x \chi\, y$) when the following property holds:

$$x \chi\, y \stackrel{def}{=} \forall i \in I, \forall o \in O, ((i \preceq x \wedge y \preceq o) \Rightarrow (i \preceq o)) \ \wedge \ ((i \preceq y \wedge x \preceq o) \Rightarrow (i \preceq o))$$

The reverse relation, *incompatibility*, is the property which formalizes the fact that two nodes (either internal, input or output) can not be statically scheduled within the same piece of atomic code: if two actions are not compatible (for instance $i \preceq x$, $y \preceq o$ and $i \not\preceq o$), it is possible to feedback o to the input i without introducing a combinatorial loop. Thus, setting x and y together into the same piece of code would improperly make this feedback impossible.

The goal is to find classes of pairwise compatible actions in A. Note that compatibility is not a solution, since it is not an equivalence relation. It is symmetric, reflexive, but not transitive in general: for example, in Figure 6, $f \chi\, j$ and $f \chi\, b$ hold but not $j \chi\, b$.

Moreover, not any equivalence relation $\simeq$ included in χ is a solution: the atomic blocks must be made of pairwise compatible actions, but they must also be schedulable with respect to $\preceq$, without introducing any extra dependencies between inputs and outputs.

In other terms, we are looking for an equivalence relation plus a (partial) order over the classes. This is strictly equivalent to search a *preorder* (reflexive, transitive, not symmetric relation) over the set of actions.

Definition 3 (Optimal Static Scheduling (OSS)) A static schedule is a relation $\mathrel{-} \subseteq A \times A$ such that:

- (SS-0) $\mathrel{-}$ is a *preorder* (reflexive, transitive), and we note $\simeq$ the underlying equivalence relation ($x \simeq y \ \Leftrightarrow \ x \mathrel{-} y \ \wedge \ y \mathrel{-} x$),
- (SS-1) $x \preceq y \Rightarrow x \mathrel{-} y$ (it contains all the dependency constraints),
- (SS-2) $\forall i \in I, \forall o \in O, \ \ i \mathrel{-} o \ \Leftrightarrow \ i \preceq o$ (the preorder strictly maps the dependency on input/output pairs).

Moreover, a static schedule is called optimal (OSS) if it satisfies the following property:

- (SS-3) $\simeq$ is maximal (i.e. it has a minimal number of classes).

Property 1 The main property following the definition SS-0 to SS-2 is that the underlying equivalence $\simeq$ implies the compatibility relation:

– (SS-prop) $x \simeq y \Rightarrow x \chi y$

Proof Let $\dashv$ be a static schedule. Suppose that there exist some actions x, y, input i and output o such that $x \simeq y$, $i \preceq x$, $y \preceq o$ and $i \not\preceq o$, by SS-1, $i \dashv x$ and $y \dashv o$, then, since (in particular) $x \dashv y$, then $i \dashv o$ and, consequently, by SS-2 $i \preceq o$ must hold, which is a contradiction.

Note that the dependency $\preceq$ is trivially a non-optimal static schedule: choosing this solution corresponds to the white-boxing approach (one computation block for each atomic action).

3.2 Theoretical Complexity of OSS

An equivalent definition of the problem, presented in terms of graphs and clusters, has been proposed in [12]. Authors have shown that it is NP-complete, through a reduction of the *k-cliques partition* problem, also known as *clique cover* [5].

Definition 4 (Minimal Clique Cover (Richard Karp, 1972)) A *clique* in a non-oriented graph G, is a subset of vertices that are all connected pairwise. The MCC problem consists in finding a partition of the vertices into a (minimal) number of cliques.

We briefly present an alternative encoding, thereafter called X-encoding ("cross"-encoding), which establishes the same result. This is motivated as it helps to better understand why the problem is theoretically hard, while being simple on small but typical systems such as the one given in Figure 6.

We show that any *minimal clique cover* problem can be linearly encoded into a OSS instance. First, we reformulate the problem in terms of relations: given a finite set L and a symmetric relation $\leftrightarrow$ (i.e. a non oriented graph), find a maximal equivalence relation $\simeq$ included in $\leftrightarrow$ (i.e. with a minimal number of classes, called cliques in the original formulation).

This problem has indeed many similarities with OSS since OSS mainly consists in finding a maximal equivalence included in a symmetric relation (the compatibility). However, it is not trivial that compatibility relations can be as general as symmetric relations.

X-encoding Let $G = (L, \leftrightarrow)$ be the data of a MCC problem. We build a OSS instance with an extra set of nodes X: $(A = L \uplus X, I = O = X, \preceq)$ (i.e. each extra variable is both an input and an output), in the following way:

1. For each $x \in L$, we introduce 4 extra variables io^1_x, io^2_x, oi^1_x and oi^2_x that are *both* input and output. Each local variable x and its extra input/output variables are related by the dependency relation in a "X" (cross) shape, as presented on the left side of Figure 9.
2. For each pair x, y such that $x \leftrightarrow y$, we add 8 dependencies by connecting each input of x to each output of y and vice versa, as shown on the right side of Figure 9.

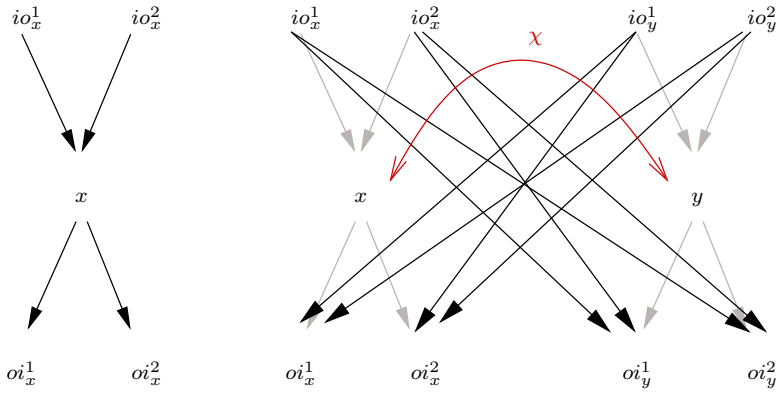


Fig. 9 The X-encoding, isolated node (left), and related pair (right)

The X-encoding ensures that each additional node (io_x^1,io_x^2, oi_x^1 and oi_x^2) is incompatible with any other variables. For instance, an extra variable io_x^1 is incompatible:

- with any other extra variable of type io or oi, since it can be fed back to any of them,
- with any local node $y \neq x$, since it can be fed back either to io_y^1 or io_y^2,
- with the associated local variable x since it can be fed back to the other extra input io_x^2.

The same reasoning applies for variables of type oi.

The dependencies added for each pair $x \leftrightarrow y$ enforce that x is compatible with y: all outputs of y depend on all inputs of x and reciprocally. On the contrary, if $x \not\leftrightarrow y$, no extra dependencies are introduced and $x \not\chi y$: any output of x can be fed back to any input of y and reciprocally.

Finally, we have $x \chi y \quad \Leftrightarrow \quad x \leftrightarrow y$. Note that it establishes that a compatibility relation can be an arbitrary symmetric relation.

Theorem 1 (OSS is NP-hard) *OSS is as difficult as MCC, since any instance of MCC can be linearly reduced to an particular instance of OSS.*

Let $G = (L, \leftrightarrow)$ be an instance of the the MCC problem, and $X_G = (L \cup X, \preceq)$ the instance of the OSS problem obtained by X-encoding.

Firstly, X-encoding is trivially linear according to the size of the MCC instance. Secondly, an optimal solution for X_G can be (linearly) deduced for any optimal solution of G, and conversely, as proven in the sequel.

1. Let $\simeq$ be a solution of the clique cover problem over $(L, \leftrightarrow)$ and consider the relation R over $(L \cup X, \preceq)$ which is the *transitive closure* of the relation $\simeq \cup \preceq$. This relation:
   - is a preorder (SS-0),
   - by definition, it contains $\preceq$ (SS-1),
   - it does not add any extra dependency between inputs and outputs (SS2).

Proof Firstly, note that "inputs" (nodes of type io), are minimal by construction for both $\simeq$ and $\preceq$, and then, the transitive closure cannot introduce any relation between two nodes of type io. The same reasoning holds for "outputs" (type oi) which are maximal by construction. As a consequence, the transitive closure necessarily crosses internal nodes.
Secondly, since by construction $x \not\preceq y$ for any pair of internal nodes, $x\,R\,y$ is necessarily due to the transitive closure of $\simeq$, which is, by definition already transitive, thus: $x\,R\,y \;\Leftrightarrow\; x \simeq y$.
Now, suppose that there exists an input/output pair such that $i\,R\,o$ and $i \not\preceq o$; then one can find two internal nodes x, y such that $i \preceq x$, $x\ R\ y$, and $y \preceq o$. But $x\ R\ y \;\Leftrightarrow\; x \simeq y$, and, since $\simeq$ is a solution of the clique cover, $x \simeq y \;\Rightarrow\; x \leftrightarrow y$. Finally, we have $i \preceq x$, $y \preceq o$ and $x \leftrightarrow y$, and then, because of the X-encoding construction (left side of Figure 9), $i \preceq o$, which is absurd.

2. Let - be a solution of the X-encoded problem over $(L \cup X, \preceq)$ and $\simeq$ the associated equivalence. Since $\simeq \subseteq\ \chi$ and χ only concerns internal nodes, then $\simeq$ is also an equivalence over L and thus, a solution of the clique-cover problem.
3. Moreover, an optimal solution for one problem necessarily gives an optimal solution for the other: suppose, for instance, that $\simeq$ is an optimal solution of the clique cover, and that the transitive closure R of $\simeq \cup \preceq$ is not an optimal scheduling. Then it exists a strictly better scheduling - from which we can derive, by (2), a solution of the clique cover which is strictly better than the optimal: this is absurd.

A main interest of this encoding is that it suggests that, even if OSS is proven to be at least as complex as MCC, the instances of OSS that are supposed to be computationally hard are far from what is a typical data-flow program: in order to fit the general case, we have to consider programs where the number of inputs/outputs is larger than the number of internal nodes, an uncommon situation in practice.

Our purpose is to establish that the complexity of OSS is strongly related to the number of inputs/outputs, and the structure of their dependencies.

4 Input/Output Analysis

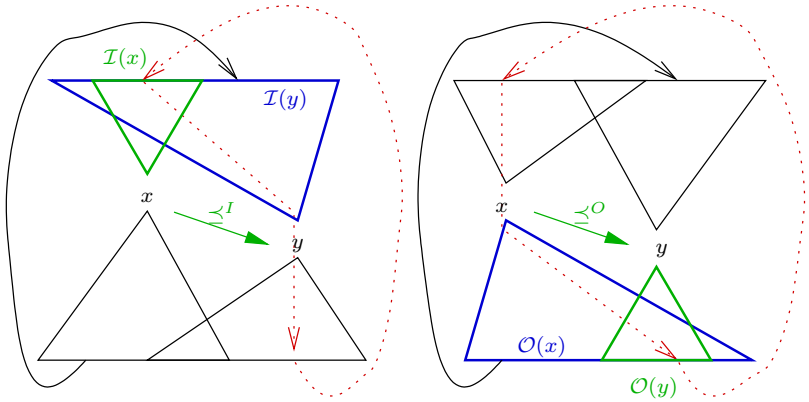
This section shows that the OSS problem can be reformulated, and in some sense *simplified*, by analyzing the relations between input and output.

4.1 Input Saturation and Output Saturation

Let $(A, I, O, \preceq)$ be a data-flow network.

Comparing the sets of inputs (or outputs) of two nodes may give information on how they can be statically scheduled without forbidding valid feedbacks. Let $\mathcal{I}(x)$ be the input variables on which x depends on. Let $\mathcal{O}(x)$ be the output variables depending on x.

Figure 10 illustrates the possible relations between two actions x and y that are not related by $\preceq$. Top (bottom) triangles represent the set of predecessors (successors) of an action according to $\preceq$; top (bottom) lines correspond to inputs (outputs). As shown in this figure:



(a) $x \preceq^I y$ (b) $x \preceq^O y$

In both case: x may be "fed-back" to y, but y cannot

Fig. 10 Preorders $x \preceq^I y$ and $x \preceq^O y$ give extra information on feasible scheduling.

- (a) if the inputs $\mathcal{I}(x)$ of x are included in the inputs $\mathcal{I}(y)$ of y, then any feedback from an output of y to some input of x creates a combinatorial loop. As a consequence, it is never the case that y *should be computed* before x. In other words, x can always be computed before y. We write $x \preceq^I y$.
- (b) the same reasoning holds with the (reverse) inclusion of outputs: if $\mathcal{O}(x) \supseteq \mathcal{O}(y)$ then x can always be computed before y. We write $x \preceq^O y$.

These relations $\preceq^I$and $\preceq^O$, that are trivially preorders, are called respectively the *input* and *output* saturations of the relation $\preceq$. They are solutions of SS:

- (SS-1) they both include $\preceq$, since $x \preceq y$ implies $\mathcal{I}(x) \subseteq \mathcal{I}(y)$ and $\mathcal{O}(x) \supseteq \mathcal{O}(y)$,
- (SS-2) they do not introduce any extra input/output dependency, since $\mathcal{O}(i) \supseteq \mathcal{O}(o)$, and, in particular, $o \in \mathcal{O}(o)$, we have $i \preceq o$ (similarly for the inclusion of input).

It follows that, from the basic dependency relation (which is a non optimal solution of SS), one can derive another solution which is still non optimal, but better than the original.

Figure 11 illustrates the computation of the relation $\preceq^O$:

- on the right-hand side, the subset of outputs is associated to each node; outputs inclusion allows to relate nodes that where not related before (dashed arrows, transitivity and symmetry is kept implicit),
- the right-hand side illustrates the whole $\preceq^O$ relation, given as a set of partially ordered classes. This preorder has 4 classes, and thus, it is not an optimal static scheduling (Figure 6 gives two solutions with only 2 classes). However, it is better than the trivial solution $\prec$, where the underlying equivalence is identity (with 9 classes).

Regarding previous definitions, we have build $\preceq^I$ and $\preceq^O$ starting from the data-dependence relation $\preceq$ which is already a valid solution to the static scheduling problem.

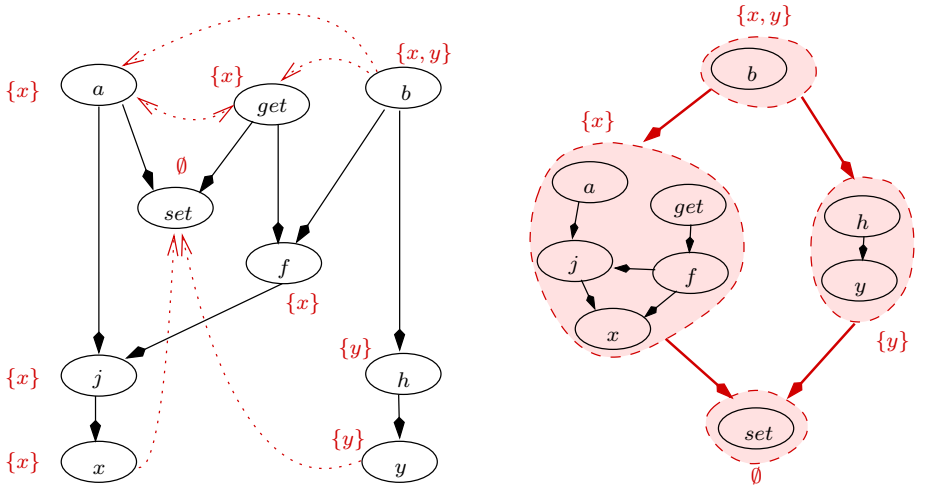


Fig. 11 Computing outputs reverse inclusion (left), gives a preorder $\preceq^O$ with 4 classes (right).

In fact, the same reasoning holds when starting from any preorder which is a solution of SS.

Definition 5 (Input and output saturation) Given a solution - of the SS problem, we define its input (or output) function, saturation preorder and equivalence:

$$\begin{aligned}
\mathcal{I}^{\precsim}(x) &= \{i \in I \,, i \text{-} x\} \\
x \text{-}^I y &\Leftrightarrow \mathcal{I}^{\precsim}(x) \subseteq \mathcal{I}^{\precsim}(y) \\
x \simeq^I y &\Leftrightarrow \mathcal{I}^{\precsim}(x) = \mathcal{I}^{\precsim}(y) \\
\mathcal{O}^{\precsim}(x) &= \{o \in O \,, x \text{-} o\} \\
x \text{-}^O y &\Leftrightarrow \mathcal{O}^{\precsim}(x) \supseteq \mathcal{O}^{\precsim}(y) \\
x \simeq^O y &\Leftrightarrow \mathcal{O}^{\precsim}(x) = \mathcal{O}^{\precsim}(y)
\end{aligned}$$

We list the main properties of the saturation preorders. We provide proofs only for the input-saturation -^I (the case of -^O is dual):

Property 2 -^I and -^O are solutions of SS.

Proof (for -^I*)*

- (SS-1) since - is itself a solution of SS, $x \preceq y \Rightarrow x \text{-} y$, and $x \text{-} y \Rightarrow \mathcal{I}^{\precsim}(x) \subseteq \mathcal{I}^{\precsim}(y)$
- (SS-2) if $\mathcal{I}^{\precsim}(i) \subseteq \mathcal{I}^{\precsim}(o)$ then, since in particular $i \in \mathcal{I}^{\precsim}(i)$, we have $i \text{-} o$, which implies $i \preceq o$ since - satisfies SS-3

The same reasoning applies for -^O.

Property 3 for any SS solution, input-saturation equivalence is identical to compatibility on outputs, and output-saturation equivalence equals compatibility on inputs:

$$\begin{aligned}
\forall o, o' \in O \quad o \simeq^I o' &\Leftrightarrow o \chi o' \\
\forall i, i' \in I \quad i \simeq^O i' &\Leftrightarrow i \chi i'
\end{aligned}$$

Proof (for $\simeq^I$) For all output o, o',

- $o \simeq^I o' \Rightarrow o \chi\, o'$, because of SS-prop,
- in order to prove the converse, since χ is symmetric, it is enough to show that $o \chi\, o' \Rightarrow o \text{ - }^I o'$.
  If $o \not\text{-}^I o'$, then it exists $i \in I$ such that $i \text{ - } o$ and $i \not\text{-} o'$, from SS-2, we have also $i \preceq o$ and $i \not\preceq o'$, and then o and o' are not compatible.

The same reasoning applies for $\simeq^O$.

Property 4 In any *optimal solution* of SS, two inputs or two outputs that are compatible are necessarily in the same class.

Proof (for output pairs)

- Suppose that - is an optimal solution of SS, where two compatible outputs are not equivalent ($o \not\simeq o'$) then compute its input-saturation $\simeq^I$.
- From the properties of $\simeq^I$, $o \simeq^I o'$, and then $\simeq^I$ and $\simeq$ are different. Moreover $\simeq^I$ is greater or equal ($\simeq^I \supseteq \simeq$) than $\simeq$.
- Finally, $\simeq^I$ has strictly less classes than the supposed optimal solution, which is a contradiction.

The proof is similar for input pairs.

4.2 Input/Output Saturation

Saturation "works" only once (applying it twice provides no further changes): let - be a SS solution and -^I its input-saturation, then $i \text{ -}^I x \Leftrightarrow i \text{ - } x$. In other words, inputs according to -^I are exactly inputs according to -. The same applies for the output-saturation.

However, performing input-saturation followed by output-saturation (or vice-versa) may lead to further improvements.

Figure 12 illustrates the input saturation of the output saturation of $\preceq$ (already presented Figure 11). Inputs according to $\preceq^O$ are computed (left), which leads to a preorder with only 2 classes (right). For this particular case, we can state that this is an optimal solution: it corresponds to the optimal solution already presented in right-hand side of Figure 6.

In other terms, by computing output-saturation followed by input-saturation, we can obtain a solution which is better than either $\preceq^I$ and $\preceq^O$ (in fact optimal for this particular simple example). In the sequel, we precisely define this combined saturation and study its main properties.

Definition 6 (Input/output preorder) Let - be a solution of SS, we define its input/output saturation -^{I_O}, and equivalence $\simeq^{I_O}$, as the input-saturation of its output-saturation:

$$\mathcal{I}^{\preceq}_{\mathcal{O}}(x) = \{i \in I \,, i \text{ -}^O x\}$$
$$x \text{ -}^{I_O} y \Leftrightarrow \mathcal{I}^{\preceq}_{\mathcal{O}}(x) \subseteq \mathcal{I}^{\preceq}_{\mathcal{O}}(y)$$
$$x \simeq^{I_O} y \Leftrightarrow \mathcal{I}^{\preceq}_{\mathcal{O}}(x) = \mathcal{I}^{\preceq}_{\mathcal{O}}(y)$$

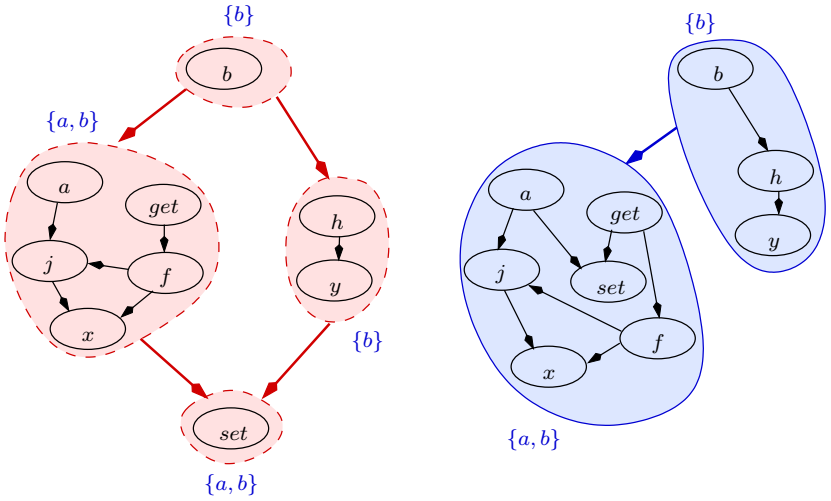


Fig. 12 Computing input saturation of $\preceq^O$

This preorder inherits from the properties of both saturations:

1. It is a solution of SS,
2. it meets the compatibility relation on inputs ($i \chi\, i' \Leftrightarrow i \simeq^{I_O} i'$),
3. it meets the compatibility relation on outputs ($o \chi\, o' \Leftrightarrow o \simeq^{I_O} o'$).

Moreover, it has a new property on input/output pairs:

Property 5 $\simeq^{I_O}$ meets compatibility on $I \times O$:

$$\forall i \in I, \forall o \in O, i \chi\, o \;\Leftrightarrow i \simeq^{I_O} o$$

Proof – $i \simeq^{I_O} o$ implies $i \chi\, o$ from SS-prop.
– $i \chi\, o \Rightarrow i \preceq o$ by definition, $i \preceq o \Leftrightarrow i \text{ - }^{I_O} o$ from SS-2.
– $i \chi\, o \Rightarrow o \text{ - }^{I_O} i$, because, if $o \not\text{-}^{I_O} i$ it exists i' such that (a) $i' \text{ - }^{O} o$ and (b) $i' \not\text{-}^{O} i$; from (a) and SS-2 $i' \text{ - } o$ and from (b) $\exists o'\ i \text{ - } o' \ \wedge\ i' \not\text{-}\ o'$, thus, from SS-2, i and o are not compatible.

The consequence of these properties is that, in any optimal solution of the SS problem, any pair of compatible nodes in $I \cup O$ are necessarily in the same equivalence class.

Moreover these equivalence classes on $I \cup O$ can be computed from any known solution of SS, in particular from the trivial "worst" solution $\preceq$. We note $\mathcal{I}_\mathcal{O}$ and $\preceq^{I_O}$ the input-output function and saturation of $\preceq$.

In other words, there is no choice for "gathering" inputs and outputs in an optimal solution. Moreover, computing $\mathcal{I}_\mathcal{O}$ gives a lower bound to the number of classes: a class containing an input and/or an output is mandatory in any valid solution. However, several problems remain to be addressed: where to put local nodes in an optimal solution? When are additional classes necessary? How many additional classes are necessary?

Our purpose in the sequel is to precise and solve these problems. We must also try to identify simple cases for which the optimal solution can be given in polynomial time from the "hard" ones, that fall into the NP-hard theoretical complexity.

4.3 Static Scheduling as a Mapping in the Input Power-set

In this section, we show that solving (O)SS is equivalent to finding a (minimal) mapping of the actions into the power-set of inputs [7]. Consider a mapping $\mathcal{K} : A \mapsto 2^I$, which satisfies the following properties:

- (KI-1) $\forall x \in I \cup O,\ \ \mathcal{K}(x) = \mathcal{I}_{\mathcal{O}}(x)$
- (KI-2) $\forall x, y,\ \ x \preceq y \ \Rightarrow\ \mathcal{K}(x) \subseteq \mathcal{K}(y)$

We call the problem of finding a mapping satisfying these properties the *Keys as inputs subsets* encoding (or KI-enc). Obviously, the dual problem of *Keys as outputs* is equivalent. We show that this problem solves the SS one, in the sense that a solution of KI-enc directly leads to a solution of SS.

Proposition 1 *KI-enc solves SS*

This proposition states that any solution of KI-enc leads to a solution of SS.

Proof Indeed, the mapping $\mathcal{I}_{\mathcal{O}}$ is a trivial solution of KI-enc, which provides, through keys inclusion, a solution of SS which is simply $\preceq^{I_O}$. The result is nonetheless more general: let $\mathcal{K}$ be any solution of KI, then the preorder $x \,\text{-}\, y \Leftrightarrow \mathcal{K}(x) \subseteq \mathcal{K}(y)$ is a solution of SS:

- SS-1 follows from KI-2,
- SS-2 follows from KI-1 and the properties of $\mathcal{I}_{\mathcal{O}}$.

Proposition 2 *SS solves (and is improved by) KI-enc*

This proposition states that, from any solution of SS, one can build a solution of KI-enc, which is itself a better (greater according to relation inclusion) static scheduling.

Proof Suppose that - is a solution of SS, apply output and then input saturation, the corresponding $\mathcal{I}_{\widetilde{\mathcal{O}}}^{\preceq}$ is a solution of KI-enc, and, moreover the corresponding preorder is equal or better (i.e. included) than -.

These properties are interesting for the search of optimal solutions, that is, preorders with a minimum number of classes. They imply that (1) any solution of KI-enc gives a solution of SS, (2) from any solution of SS, one can build a *better or identical* solution of KI-enc. In other terms, the KI-enc formulation has strictly less solutions than SS, but it does not discard *any optimal solution*.

4.4 Computing the KI-enc System

Now, we reformulate the KI-enc problem into a more computational problem. First of all, it is not necessary to compute explicitly the dependency order $\preceq$. We suppose that this order is given implicitly by a direct acyclic graph $\rightarrow$, deduced linearly from the wires in the original data-flow program (e.g. given in SCADE or SIMULINK). Finally, we consider a graph $(A, I, O, \rightarrow)$:

- where A is a set of n nodes (actions),

[7] All the definition and properties we give can be replayed, by computing instead a mapping into the power-set of outputs.

- I is the subset of n_i inputs,
- O is the subset of n_o outputs,
- $\rightarrow \in A \times A$ is a dependency graph of p arcs, whose transitive closure is the dependency (partial) order $\preceq$. We do not require this graph to be minimal: the only interesting property is that is size p is linear with respect to the size of the original program.

We build a system of (in)equalities with a variable K_x for each $x \in A$ such that:

$$K_x = \mathcal{I}_{\mathcal{O}}(x) \qquad \text{for } x \in I \cup O$$

$$\bigcup_{y \rightarrow x} K_y \subseteq K_x \subseteq \bigcap_{x \rightarrow z} K_z \quad \text{otherwise}$$

Where $\bigcup$ and $\bigcap$ are the natural n-ary extensions of union and intersection, that is, they map 0-argument to their respective neutral element: $\bigcup_{\emptyset} = \emptyset$, and $\bigcap_{\emptyset} = I$.

Complexity analysis The complexity of building the KI-enc system from a graph $(A, I, O, \rightarrow)$ is mainly due to the computation of $\mathcal{I}_{\mathcal{O}}$:

- computing the output function $\mathcal{O}$ can be done by assigning to each node a set variable O_x, and then by visiting once, in a bottom-up topological order, the n nodes and p arcs of the dependency graph. For each arc $x \rightarrow y$, a set union must be made ($O_x := O_x \cup O_y$) whose cost is theoretically in $n_o \cdot \log n_o$ (negligible, in practice, for small values of n_o when sets are encoded as bit-vectors). The global cost is of the order $n + p \cdot n_o \cdot \log n_o$.
- given $\mathcal{O}$, the $\mathcal{I}_{\mathcal{O}}$ of each node x can be computed by comparing $\mathcal{O}(x)$ to $\mathcal{O}(i)$ (cost in $n_o \cdot \log n_o$), if it is included, i is added to the known $\mathcal{I}_{\mathcal{O}}$ of x (cost in $\log n_i$). The cost is globally of the order $n \cdot n_i \cdot \log n_i \cdot n_o \cdot \log n_o$,

Finally, the system can be built in (roughly) $z \cdot m^2 \cdot (\log m)^2$, where $z = n + p$ characterizes the size of the graph, and $m = max(n_i, n_o)$ characterizes the size of its interface.

If we isolate the cost of set operations ($B(m)$ for binary ones, and $A(m)$ for insertion), we obtain the expression $z \cdot m \cdot B(m) \cdot A(m)$ which gives a more interesting information in practice: the cost is mainly the product of the size of the graph by the size of its interface and the cost of set operations.

Iterating SAT searching The system of inequalities is trivially a SAT problem, since a subset of I can be represented by a vector of n_i Boolean values:

- each variable K_x can be encoded by a vector of Boolean variables $[K_x^1, \cdots, K_x^{n_i}]$
- each constant $\mathcal{I}_{\mathcal{O}}(x)$ can be replaced by the corresponding vector of n_i bits,
- $\bigcup$ is the bitwise logical or, $\bigcap$ is the bitwise logical and.

As we have shown in section 4.2, the system outlines the fact that some classes are mandatory: the ones appearing in equalities, that is, the $\mathcal{I}_{\mathcal{O}}$ of inputs and outputs. We call them the c mandatory classes $M_1, ...M_c$. An optimal solution can be obtained by iterating SAT-solver calls:

- search a solution with $c + 0$ classes, by adding, for each local variable x, the constraint $\bigvee_{j=1}^{j=c}(K_x = M_j)$, and calling a SAT solver,

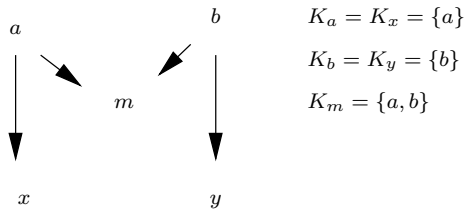


Fig. 13 A simple "M" shape, where $I = \{a, b\}$ and $O = \{x, y\}$.

- if it fails, search for a solution with $c + 1$ classes, i.e., introduce a new *vector of* n_i *free variables* S_1 and add this variable vector as a new possible value for each local x:
$$\bigvee_{j=1}^{j=c} (K_x = M_j) \vee (K_x = S_1)$$
- if it fails, introduce another extra variable S_2, and so on.

However, before applying an iterative search which may be costly, it is interesting to study more precisely the system itself and its sources of complexity.

4.5 Simplifying the KI-enc System

Lower and higher bounds For each variable, we define its lower and upper value:

- $k_x^\perp = k_x^\top = \mathcal{I}_\mathcal{O}(x)$ for $x \in I \cup O$
- $k_x^\perp = \bigcup_{y \longrightarrow x} k_y^\perp$ and $k_x^\top = \bigcap_{x \longrightarrow z} k_z^\top$ otherwise.

It is easily to show that $k_x^\top$ equals $\mathcal{I}_\mathcal{O}(x)$: setting all variables to their upper bound corresponds to the already known solution $\preceq^{I_O}$. It corresponds to the heuristic "schedule as late as possible" and we call this solution $\mathcal{K}^\top$. Dually, choosing $K_x = k_x^\perp$ for each variable is also a solution which corresponds to the heuristic "schedule as soon as possible" and we note this solution $\mathcal{K}^\perp$. These two solutions are not comparable nor optimal in general. However, if it appears that all $k_x^\top$ (respectively $k_x^\perp$) are mandatory keys (i.e. keys of some input or output), we can already conclude that $\mathcal{K}^\top$ (respectively $\mathcal{K}^\perp$) is an optimal solution. That is, no new class have been introduced.

In some cases, computing the bounds may help to discover new mandatory classes: whenever $k_x^\perp = k_x^\top$, the inequality becomes an equality and the class becomes mandatory even if it is not the class of any input nor output. This is illustrated in Figure 13: the local node m must be computed alone in a class that should be scheduled after the class of a and the class of b.

Bounds reduction Upper and lower bounds can be introduced into the inequalities, whose general form is then:

$$k_x^\perp \cup \bigcup_{y \longrightarrow x} K_y \subseteq K_x \subseteq k_x^\top \cap \bigcap_{x \longrightarrow z} K_z$$

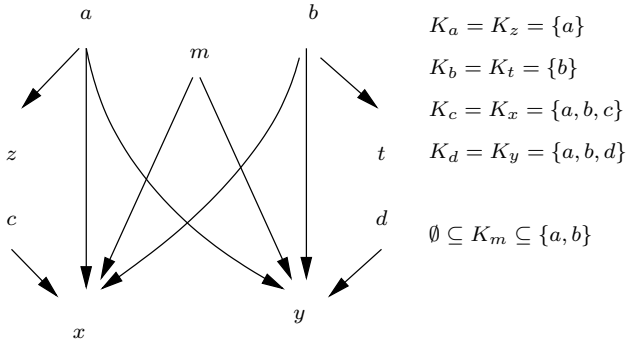


Fig. 14 A "M/W" shape, where $I = \{a, b, c, d\}$ and $O = \{x, y, z, t\}$.

For any $y \rightarrow x$ such that $k_y^\top \subseteq k_x^\perp$, the constraint $K_y \subseteq K_x$ is redundant and can be removed; similarly for any $x \rightarrow z$ such that $k_x^\top \subseteq k_z^\perp$. This simplification is called the (redundant) bounds reduction. With this reduction, it may appear that some variables become bounded by constants:

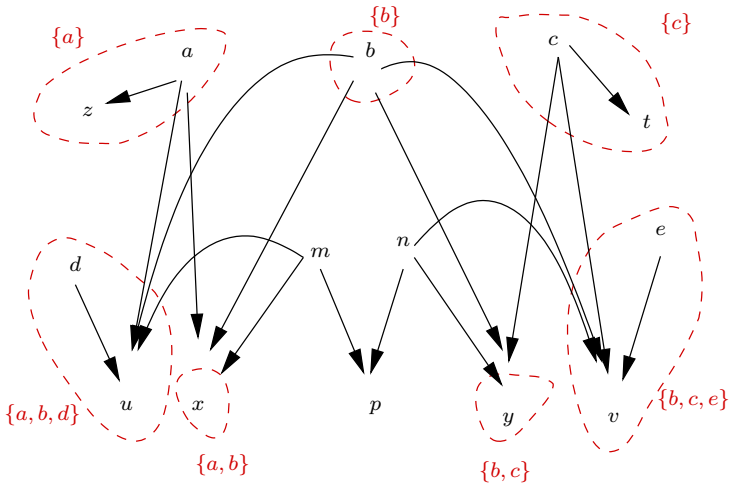
$$k_x^\perp \subseteq K_x \subseteq k_x^\top$$

Whenever $k_x^\perp \neq k_x^\top$ for some x, there are several choices for the position of x within the static schedule, that may lead or not to an optimal solution. Figure 14 shows a small example with a "M/W" shape where the position of the local variable m is bounded by two constants. In this case, no extra class is necessary if the intersection of all the intervals contains a mandatory class. In this example, the answer is trivial since we have only one interval. It includes both the mandatory classes $\{a\}$ and $\{b\}$ and one can choose either $K_m = \{a\}$ or $K_m = \{b\}$ to obtain an optimal solution.

As soon as there are more than 3 such variables, we fall in a case that is equivalent to the one of the X-encoded problem: n independent internal variables have to be organized in a minimum number of classes, according to their bounds.

The problem is even more complex when the bounds are not constant: the position of some local variable may depend on the position of another variable and so on. Figure 15 illustrates this case by generalizing the "M/W" shape. Several optimal choices can be made, that do not require extra classes. For instance, m and m can be gathered with n, that is, $K_m = K_n = K_p = \{b\}$. Another possible solution is $K_m = \{a\}, K_n = \{b\}, K_p = \{a, b\}$ and they are many more. But these optimal solutions cannot be obtained by considering variables one by one. For instance, $K_m = \{a\}$ is locally a good choice as well as $K_n = \{c\}$; but once these two choices have been made, there is no solution for p other than $K_p = \{a, c\}$ and this forces to introduce an extra class. When several variables are related, they must be considered "all together", making the problem computationally hard.

As a conclusion, the proposed encoding may, in some cases, directly give solutions that are proven optimal and, otherwise, it exhibits the parts of the systems that are *real* sources of complexity. This appears when the system still contains inequalities whose lower and upper bounds are different and involve other variables.



$$\emptyset \subseteq K_m \subseteq \{a, b\} \cap K_p$$
$$\emptyset \subseteq K_n \subseteq \{b, c\} \cap K_p$$
$$K_m \cup K_n \subseteq K_p \subseteq \{a, b, c, d, e\}$$

Fig. 15 A generalized "M/W" shape, where $I = \{a, b, c, d, e\}$ and $O = \{x, y, z, t, u, v\}$.

5 Experimentation

We have developed a prototype implementation in OCAML [11] to experiment the principles presented previously. Two kind of examples have been considered: a collection of relatively small programs taken from the SCADE library and two real-size industrial applications from Airbus. Although not particularly optimized (e.g., all set operation are implemented using functionals maps provided by the standard library), the implementation being implemented with the standard OCAML library), the prototype requires respectively 0.15, 0.02 and 0.2 seconds on a laptop (CoreDuo II, 2.8Gh, Mac OS X) to treat the whole three benchmarks presented here. The source code used for this experiment is given in appendix A [8]. It is concise enough to be fully provided, which makes it available for most users to experiment the method.

For every SCADE (or LUSTRE) code, the dependency information is extracted from an input file and the KI-system is build. Then, the tool checks whether:

- the KI-system is only made of equalities, in which case the system is solved by simple propagation of facts. We call this the *trivial* case.
- it still contains inequalities but either all the upper bounds or all the lower bounds are mandatory (that is, they appear in some equality). In this case, either $\mathcal{K}^\top$ or $\mathcal{K}^\bot$ are proven to be optimal solutions. The system is called *solved*. It corresponds to the situation where a simple heuristic — always compute as soon as possible or always compute as late as possible — gives an optimal solution.

[8] Execution times are smaller than the one given in the EMSOFT paper due to a revised implementation.

SCADE libs	trivials	solved	others
n. of programs (223)	65	158	0
n. of classes	1	1 to 2	-
n. of in/out	2 to 5	2 to 9	-
n. of nodes	2 to 18	8 to 64	-
av. size	10	24	-

Benchmark 2	trivials	solved	others
n. of programs (27)	8	19	0
n. of classes	1	1 to 4	-
n. of in/out	2 to 10	2 to 19	-
n. of nodes	2 to 29	9 to 48	-
av. size	14	20	-

Benchmark 3	trivials	solved	others
n. of programs (125)	41	83	1
n. of classes	1 to 3	1 to 4	3 or 4
n. of in/out	2 to 14	2 to 26	4+2
n. of nodes	7 to 350	12 to 600	25
av. size	50	80	-

Results show the ranges of the number of classes (i.e. the size of the optimal solution); the programs in a particular benchmark are of various sizes, we give the ranges of the number of inputs/outputs and of internal nodes, and also the average size (in internal nodes).

Fig. 16 Experimental results: almost 100% of the programs are polynomially solved.

– in the first two cases, an optimal schedule is returned. Otherwise, the system is considered to be *complex*. It contains equalities that give a set of mandatory classes. It also contains inequalities but some upper bounds and some lower bounds are not mandatory. Thus, neither $\mathcal{K}^{\top}$ nor $\mathcal{K}^{\bot}$ are proven to be optimal. As we have presented in 4.4, the tool produces a generic Boolean formula parameterized by the number of extra classes *ec*. This formula can be expanded for $ec = 0, 1, ...$ and then handed to a Boolean solver.

Benchmark 1 (Scade Lib) The problem of modular compilation is clearly interesting for programs that are supposed to be reusable. This is why we have chosen, as first benchmark, the libraries of the SCADE tool [9]. The benchmark is made of 223 block-diagrams (we simply call them programs in the sequel), which are indeed relatively small. Not surprisingly, none of them was identified as *complex* (see Figure 16). Moreover, 91% of the programs are strict: every output depends on all inputs (this is the case for the mathematical libraries, for example). As a consequence they trivially require a single computation block. The other operators (19 over 223) are basic temporal operators (e.g. integrators) which require two blocks.

Since it seems very unlikely to find complex reusable programs, we have tried to apply the method on bigger industrial examples.

Benchmark 2 This benchmark is extracted from an industrial application from Airbus. We have extracted 27 components and sub-components of a medium sized application (about 600 atomic nodes when flattened). Indeed, performing modular compilation is not really relevant in this case since the components are not intended to be reused. But

[9] For practical reasons, we have used the libraries of the version 4.2 of SCADE.

```
{}  <= K(g1)  <=  {d, a}
{a} <= K(v10) <=  {d, a}                    K(v3)  <= K(s0)
{a} <= K(s1)  <=  {d, c, b, a}              K(v3)  <= K(v4)
{}  <= K(g0)  <=  {a}                       K(v9)  <= K(v11)
{}  <= K(v2)  <=  {a}                       K(v10) <= K(s1)
{}  <= K(v9)  <=  {d, a}                    K(v10) <= K(v11)
{a} <= K(v11) <=  {d, a}                    K(g0)  <= K(v2)
{a} <= K(v3)  <=  {c, b, a}                 K(g1)  <= K(s1)
{a} <= K(s0)  <=  {d, c, b, a}              K(g1)  <= K(v9)
{a} <= K(v4)  <=  {c, b, a}
```

Fig. 17 Inequality system of a (possibly) complex program, as it is echoed by the prototype: the constants bounds (left) and the variables constraints (right).

the goal here is to experiment the method on programs that are relatively bigger than those that can be found in libraries. It also measure the practical complexity of the method on larger programs. Finally, the conclusion is the same as in the first benchmark: all the programs are solved by a simple input/output analysis. One particular program is interesting: it has 11 inputs and 4 outputs for a total of 30 nodes, and its optimal static scheduling has 4 classes, which is the more complex static scheduling we found in our experimentation.

Benchmark 3 The last benchmark is also extracted from an Airbus application, but it is bigger than the previous one: 125 components are extracted from an application whose flat size is about 8000 nodes. Once again, the components are mostly application-specific tasks and sub-tasks than reusable operators. Thus, they are not obviously candidates for separate compilation.

The conclusion is the same as before except for one program that is not of *solved* kind (cf. column *others* in the last table of Figure 16). This program has 24 nodes, 4 inputs and 2 outputs, the minimal number of (mandatory) classes is 3, and both $\mathcal{K}^{\top}$ and $\mathcal{K}^{\bot}$ give 4 classes. The corresponding system of inequalities is simplified and the tool isolates 10 internal variables whose position is not clear. Figure 17 shows the bounds and the remaining inter-dependencies of these variables.

Looking into the program suggests that it is also of rather simple nature, but that it requires a heuristic that is not yet implemented in the prototype:

- the intersection of all the intervals is not empty ($\{a\}$);
- moreover, the class $\{a\}$ is one of the 3 mandatory classes;
- thus, setting all the variables to $\{a\}$ satisfies all the constraints without introducing any extra classes.

Conclusion Is it possible indeed to build an arbitrarily huge and *complex* data-flow network, e.g., by multiplying the "M/W" shape given in Figure 15. However, our experience is that hand-written data-flow networks are rather "simple". This is so in particular because the number of inputs/outputs of a program is relatively small with respect to the number of internal nodes.

If it is admitted that modular compilation is useful — a problem which is briefly discussed in the next section —, the proposed approach consists in using an *inexpensive* algorithm that:

- in most cases, gives an optimal solution;

– identifies the problem as being potentially "hard" and calls a third-party non-polynomial tool for solving it.

6 Discussion and Related Works

Various approaches for code generation from synchronous data-flow diagrams have been experimented in compilers.

Modular Code Generation Modular code generation, as followed by the SCADE compiler, essentially translates every stream function into a single function [3]. This way, modules can be compiled separately. Invocation conventions (to build libraries or to link synchronous code with others) are almost trivial, which simplifies traceability issues between the source and target code. It was observed that extra constraints imposed on feedback loops are well accepted by users [10]. Combined with partial inlining, all remaining causally correct programs are accepted. Note that in tools such as SCADE (but it also applies to SIMULINK), many functions are local as they are only used to structure the code: they are not exported to build libraries and are only instantiated once. Any good compiler always inlines these functions thus reducing again the set of functions that need a decomposition into classes [11]. A classical variation of this modular code generation consists in producing two functions: one to produce the outputs, and one to modify the internal state. This solution allows to compile modularly more programs (e.g. Moore machines) but it is still incomplete: for instance, it fails to compile the example given in beginning of Section 2.

This (almost) black-boxing approach to code generation was discarded in the LUSTRE compiler as it may reject causally correct programs. Programs are compiled after a full inlining of functions. This allows for the generation of efficient automata [8] but at the price of modular code generation and code size increase.

Optimal Static Scheduling The question of decomposing a data-flow graph into classes has been also studied by Benveniste & al. for the compilation of SIGNAL and was observed to be similar to the problem of code distribution [2]. It is formulated for a more expressive model of conditional scheduling constraints: a relation $x \overset{ck}{\rightarrow} y$ states that "when ck is true, y must be executed after x". A graph can be scheduled when all cycles $x_1 \overset{ck_1}{\rightarrow} x_2 \overset{ck_2}{\rightarrow} ...x_1$ are such that $\neg(ck_1 \wedge ... \wedge ck_n)$. We thus have considered the case where $ck_i = true$ for all i. Nonetheless, the question of producing an optimal solution in number of classes have not been addressed.

We have already pointed the main differences with the work of Lublinerman, Szegedy and Tripakis [12]. The optimal solution is obtained iteratively for $c = 1, 2, ...$ with a general solution based on a SAT encoding of a decomposition into c classes. Nonetheless, real programs have simple dependency relations and can be treated with a dedicated algorithm with mastered worst-case complexity. In most applications we have found, the optimal is obtained in polynomial time using the direct algorithm based on the input/output relations.

[10] This is also due to the nature of applications written in SCADE where closed loops are programmed to be robust to the insertion of a delay.

[11] Nodes can also be inlined on demand, as provided by the SCADE compiler.

Input/Output Analysis vs *Direct Boolean Encoding* Since programs are *simple* in practice, one may argue that there is no much difference in actual cost between a direct boolean encoding of the OSS problem such as [12] and a dedicated algorithm. We could expect that each SAT problem — there exist a solution with c classes — is easy to solve and the number of iterations for c to stay small.

Nonetheless the behavior, and thus, the cost, of a SAT resolution is hard to predict, whereas the presented algorithm returns in polynomial time whether the **KI-enc** is solved or not. In the small remaining cases where an enumerative search is necessary (if an optimal solution is really expected), the algorithm starts from a non trivial minimal number of mandatory classes. In several benchmarks, this minimal number of classes was greater than 1. Moreover, the symbolic representation is also better than a direct boolean encoding of the static scheduling problem.

Finally, is not clear that finding an optimal solution is central in practice. In all the examples we have treated $\mathcal{K}^{\perp}$ and/or $\mathcal{K}^{\top}$ allows to obtain either the optimal solution or a solution close to the optimal (only one extra class).

7 Conclusion

This paper addresses the static scheduling problem of a synchronous data-flow network, to be used in a compiler which generate sequential imperative code. We focus on how to decompose a data-flow network into a minimal number of classes executed atomically without restricting possible feedback loops between input and output. Though this optimization problem is intractable in the general case and can be tackled with general combinatorial methods, our experience is that real programs do not expose such a complexity. This calls for a specific algorithm able to identify programs which can be solved in polynomial time. Based on the notion of input/output properties, we build a symbolic representation (keys as inputs subsets, or KI-enc) of the problem. This representation *simplifies* the problem in the sense that it has strictly less solutions but it contains *all* optimal ones. This representation gives a non trivial lower bound on the number of classes, and two particular solutions $\mathcal{K}^{\perp}$ and $\mathcal{K}^{\top}$. It can then be checked whether $\mathcal{K}^{\perp}$ and/or $\mathcal{K}^{\top}$ are optimal or not by comparing their number of classes with the lower bound. In most real programs we have encountered, $\mathcal{K}^{\perp}$ is optimal and computed with a guaranteed polynomial complexity. Otherwise, the non trivial bound on the number of classes is used to start a combinatorial search with a SAT-solver.

Acknowledgments

We thank Léonard Gérard and Andrei Paskevich for their careful reading and comments.

References

1. A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

2. Albert Benveniste, Paul Le Guernic, and Pascal Aubry. Compositionality in dataflow synchronous languages: Specification & code generation. Technical Report R3310, INRIA, November 1997.
3. Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008. Extended version of [4].
4. Darek Biernacki, Jean-Louis Colaco, and Marc Pouzet. Clock-directed Modular Compilation from Synchronous Block-diagrams. In *Workshop on Automatic Program Generation for Embedded Systems (APGES)*, Salzburg, Austria, october 2007. Embedded System Week.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability - A guide to the Theory of NP-completeness*. Freeman, New-York, 1979.
6. Georges Gonthier. *Sémantiques et modèles d'exécution des langages réactifs synchrones*. PhD thesis, Université Paris VI, Paris, 1988.
7. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
8. N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.
9. Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.
10. E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(2), 1987.
11. Xavier Leroy. The Objective Caml system release 3.11. Documentation and user's manual. Technical report, INRIA, 2009.
12. R. Lublinerman, C. Szegedy, and S. Tripakis. Modular Code Generation from Synchronous Block Diagrams — Modularity vs. Code Size. In *ACM Principles of Programming Languages (POPL)*, 2009.
13. R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe (DATE'08)*, 2008.
14. Marc Pouzet and Pascal Raymond. Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation. In *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.
15. Pascal Raymond. Compilation séparée de programmes Lustre. Technical report, Projet SPECTRE, IMAG, juillet 1988.

A The Source Code

Below are the two Ocaml modules that implement the polynomial semi-solver of OSS. The code is short, the only lacking part being the construction of the graph from a description of data-dependences written in a file. Yet, all numbers reported in section 5 have been obtained using these two modules. Sets are represented using the standard Ocaml library of finite maps represented as balanced binary trees, thus, with logarithmic access. The module `Network` defines the basic data-types and representations for sets, the association tables for I, O, IO and KI. The module `Oss` defines the IO and KI functions. Given a network $(A, I, O, \preceq)$, the main function `main` answers whether the system is trivial, solved or complex. In the first two cases, it returns an optimal static scheduling of the network.

```
(** module Network.*)
(** Nodes, graphs and auxiliary functions *)
(** The type of nodes *)
type node = { _nm : string; _index : int; _isin : bool; _isout : bool; }

let name_of_node x = x._nm
let is_input x = x._isin
let is_output x = x._isout

(** A Map associating values to nodes *)
module NodeMap =
  Map.Make
```

```
    (struct
       type t = node
       let compare n1 n2 = compare n1._index n2._index
     end)

(** A Map associating values to strings *)
module IndexMap =
  Map.Make (struct type t = string let compare = compare end)

(** Set of nodes *)
module NodeSet =
  struct
    include Set.Make
      (struct
        type t = node
        let compare n1 n2 = compare n1._index n2._index
      end)

    let of_list (l: elt list) =
      List.fold_left (fun s e -> add e s) empty l
  end

(** A Map associating node sets to node sets *)
module NodeSetMap =
  struct
    include Map.Make
      (struct
        type t = NodeSet.t
        let compare = NodeSet.compare
      end)

    let cardinal t = fold (fun _ _ c -> c + 1) t 0

    (* Add a new entry node n for a key. If t(key) already exists, extends *)
    (* t(key) with n *)
    let add k n t =
      let p =
        try find k t
        with | Not_found -> NodeSet.empty
      in
        add k (NodeSet.add n p) t
  end


(** The type of networks *)
type t = {
  mutable nbnodes : int ; (* number of nodes *)
  mutable nodes : node list ; (* the list of nodes *)
  mutable ins : node list ; (* input nodes *)
  mutable outs : node list ; (* output nodes *)
  mutable pred : node list NodeMap.t; (* the table of predecessors *)
  mutable succ : node list NodeMap.t; (* successors *)
  mutable nodetab : node IndexMap.t; (* associating nodes to names (string) *)
}

let pred network n =
  try NodeMap.find n network.pred with | Not_found -> []
let succ network n =
  try NodeMap.find n network.succ with | Not_found -> []
(** end of module Network *)

(** Module OSS *)
(** Definition of I/O, IO and KI *)
open Network

(** Computation of inputs and outputs *)
let input_output network =
  (* traveling in a graph *)
  let travel is_in next =
    (* Memorization table *)
```

```
    let table = ref NodeMap.empty in
    let rec travelrec n =
      try NodeMap.find n !table
      with | Not_found ->
            let pl = next n in
            let acc0 =
              if is_in n then NodeSet.singleton n else NodeSet.empty in
            let l =
              List.fold_left
                (fun acc n -> NodeSet.union acc (travelrec n)) acc0 pl in
           table := NodeMap.add n l !table;
           l
    in
    List.iter (fun n -> ignore (travelrec n)) network.nodes;
    !table in

  let input_table = travel is_input (pred network) in
  let output_table = travel is_output (succ network) in
  input_table, output_table

(** Computation of io for input nodes only *)
let io_of_inputs network toutput =
  let table = ref NodeMap.empty in
  let compute_io i =
    let o = NodeMap.find i toutput in
    List.fold_left
      (fun acc i' ->
         let o' = NodeMap.find i' toutput in
         if NodeSet.subset o o' then NodeSet.add i' acc else acc)
      NodeSet.empty network.ins in

  List.iter
    (fun i -> let io = compute_io i in table := NodeMap.add i io !table)
    network.ins;
  !table

(** Computing Kmin or Kmax*)
(** stop: returns the key or raise Not_found *)
(** compose: composition ok keys (intersection or union) *)
(** neutral: neutral element of compose (all or empty) *)
(** next: connected nodes (pred or succ) *)
let travel network stop neutral compose next =
  (* Memoization table *)
  let table = ref NodeMap.empty in
  let rec travelrec n =
    try NodeMap.find n !table
    with | Not_found ->
          let l =
            try stop n
            with | Not_found ->
                  let pl = next n in
                  let acc0 = neutral in
                  List.fold_left
                    (fun acc n -> compose acc (travelrec n))
                    acc0 pl
          in
            table := NodeMap.add n l !table; l
  in
    List.iter (fun n -> ignore (travelrec n)) network.nodes;
    !table

(** Computation of the KI system *)
let ki network tinput tio =
  (* compute the kmin for all nodes *)
  let stop n =
    if is_input n then NodeMap.find n tio
    else if is_output n then NodeMap.find n tinput
    else raise Not_found in
  let neutral = NodeSet.empty in
  let compose = NodeSet.union in
```

```
  let kmin = travel network stop neutral compose (pred network) in

  (* compute the kmax for all nodes *)
  let neutral =
    List.fold_left (fun acc n -> NodeSet.add n acc)
      NodeSet.empty network.ins in
  let compose = NodeSet.inter in
  let kmax = travel network stop neutral compose (succ network) in
  kmin, kmax

(** Computation of Kbot, Ktop and mandatory classes *)
let mandatory network kmin kmax =
  (* build the two inverse tables associating *)
  (* set of variables to each kset *)
  List.fold_left
    (fun (kbot, ktop, mandat, nb_eqs) n ->
       let min = NodeMap.find n kmin in
       let max = NodeMap.find n kmax in
       let mandat', nb_eqs' =
         if NodeSet.equal min max
         then (NodeSetMap.add min n mandat, nb_eqs + 1)
         else (mandat, nb_eqs) in
       (NodeSetMap.add min n kbot,
        NodeSetMap.add max n ktop,
        mandat',
        nb_eqs'))
    (NodeSetMap.empty, NodeSetMap.empty, NodeSetMap.empty, 0)
    network.nodes

(** The type of the result *)
type result =
  | Trivial of int (* every constraint is an equation *)
  | SolvedBot of int (* kbot is optimal with n classes *)
  | SolvedTop of int (* ktop is optimal with n classes *)
  | Complex of int * int
        (* the number of mandatory classes + min of nb_kbot nb_ktop *)

let print_result = function
  | Trivial(i) ->
      Printf.printf "TRIVIAL WITH %d classes.\n" i
  | SolvedBot(i) ->
      Printf.printf "SOLVED (bot) with %d classes.\n" i
  | SolvedTop(i) ->
      Printf.printf "SOLVED (top) with %d classes.\n" i
  | Complex(i,j) ->
      Printf.printf "OTHER with %d to %d classes.\n" i j

(** Check whether the system is trivial, solved or complex *)
let check network kbot ktop mandat nb_eqs =
  let nb_mandat = NodeSetMap.cardinal mandat in
  let nb_kbot = NodeSetMap.cardinal kbot in
  let nb_ktop = NodeSetMap.cardinal ktop in
  if nb_eqs = network.nbnodes then Trivial(nb_eqs)
  else if nb_mandat = nb_kbot then SolvedBot(nb_mandat)
  else if nb_mandat = nb_ktop then SolvedTop(nb_mandat)
  else Complex(nb_mandat, min nb_kbot nb_ktop)

(** The main function, [main net] returns either: *)
(** - the network [net] is trivial or simple + an optimal SS *)
(** - otherwise, it returns that the system is complex. *)
let main network =
  let tinput, toutput = input_output network in
  let tio = io_of_inputs network toutput in
  let kmin, kmax = ki network tinput tio in
  let kbot, ktop, mandat, nb_eqs = mandatory network kmin kmax in
  let result = check network kbot ktop mandat nb_eqs in
  Printf.printf "Complexity of the system: \n";
  print_result result
```