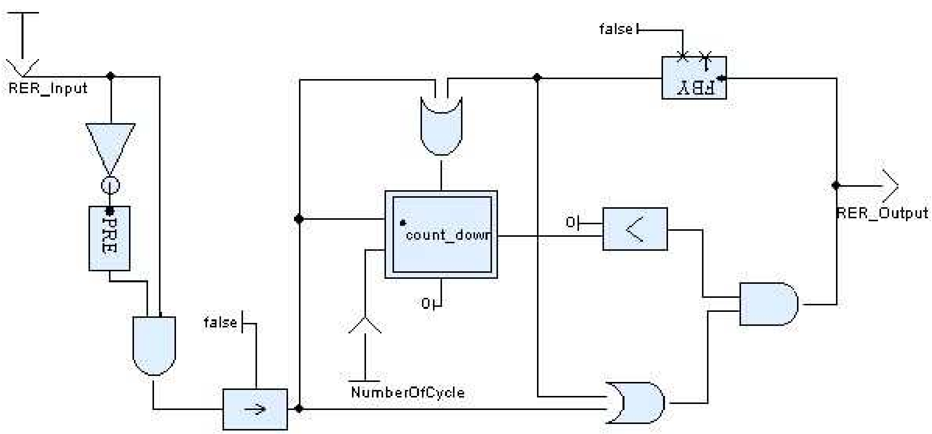
Modular Static Scheduling of Synchronous Data-flow Networks

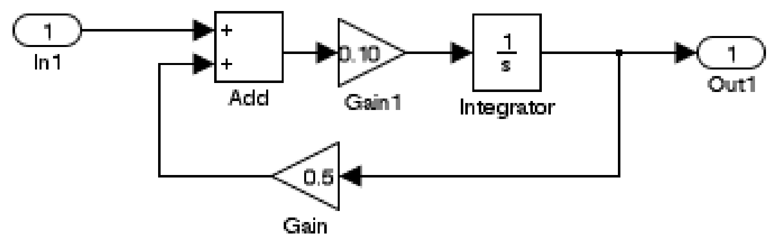
Marc Pouzet

The problem

- **Input:** a *parallel* data-flow network made of synchronous operators. E.g., LUSTRE, SCADE, SIMULINK
- **Output:** a sequential procedure (e.g., C, Java) to compute one step of the network: *static scheduling*

Examples: (SCADE and SIMULINK)





Abstract Data-flow Network and Scheduling

Whatever be the language, a data-flow network is made of:

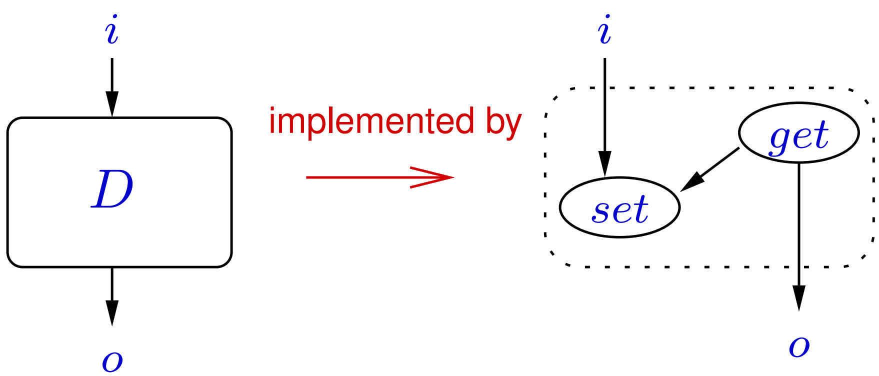
- *instantaneous* nodes which need their current input to produce their current output. E.g., combinatorial operators.

  ↪ atomic *actions*, (partially) ordered by data-dependency

- *delay* nodes whose output depend on the previous value of their input. E.g., `pre` of SCADE, $1/z$ and integrators in SIMULINK, etc.

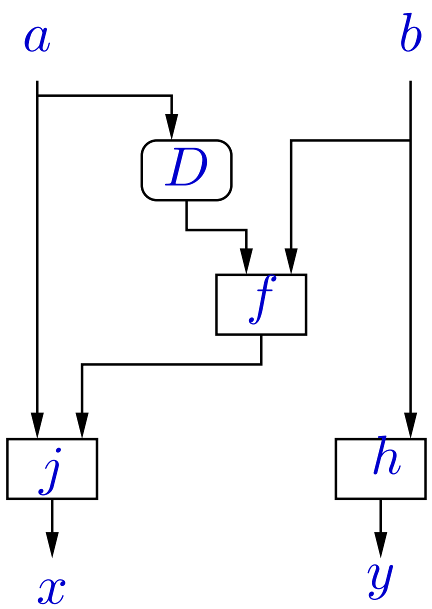
  ↪ state variables + 2 side-effect actions read (set) and update (get)

  ↪ reverse dependency (and allow feed back)



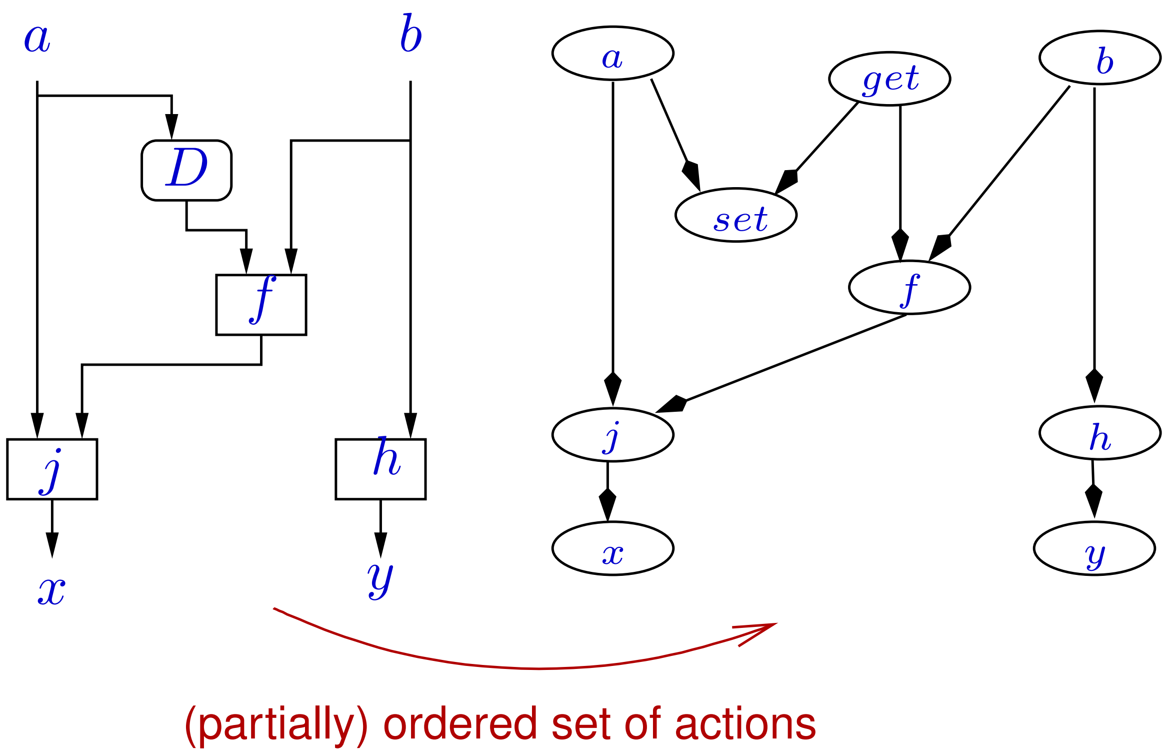
Sequential Code Generation

Build a static schedule from a partial ordered set of actions



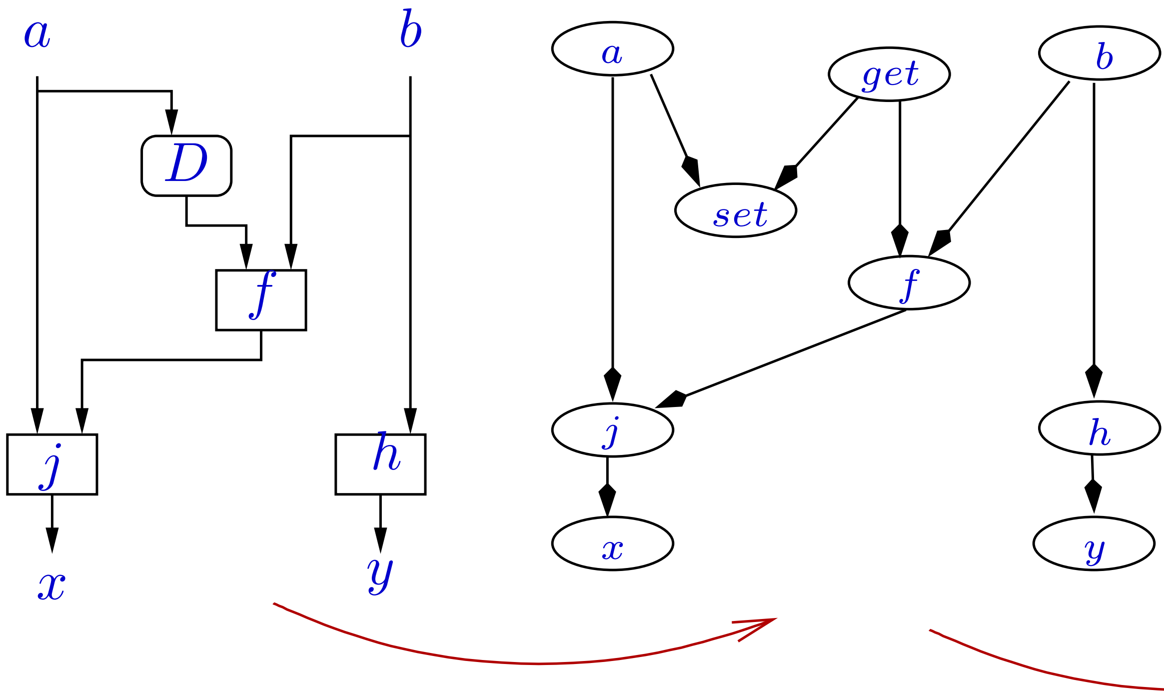
Sequential Code Generation

Build a static schedule from a partial ordered set of actions



Sequential Code Generation

Build a static schedule from a partial ordered set of actions




```
proc Step () {
  a ;
  b ;
  get ;
  f ;
  set ;
  j ;
  x ;
  h ;
  y ;
}
```

(partially) ordered set of actions

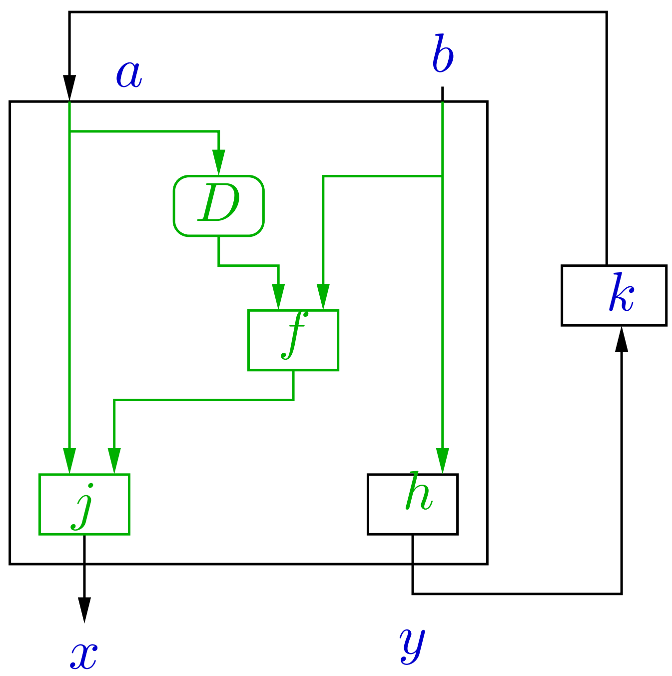
(one of the) correct sequential code

Modularity and Feedback

Modularity: a user defined node can be reused in another network

The problem with feedback loops

- this feedback is correct in a *parallel implementation*
- no *sequential single step procedure* can be used



Modularity and Feedback: classical approaches

- Black-boxing: user-defined nodes are considered as *instantaneous*, whatever be their actual input/output dependencies
  - ↪ compilation is modular
  - ↪ rejects causally correct feed-back;
  - ↪ E.g., Lucid Synchrone, SCADE, Simulink
- White-boxing: nodes are recursively *inlined* in order to schedule only atomic nodes
  - ↪ Any correct feed-back is allowed but modular compilation is lost
  - ↪ E.g., Academic Lustre compiler; on user demand in SCADE via *inline* directives.
- Grey-boxing?

Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a *(minimal) set of blocks* together with their inter-dependencies: this is called the *(Optimal) Static Scheduling Problem*
- only need to inline the *blocks dependency graph* within the caller

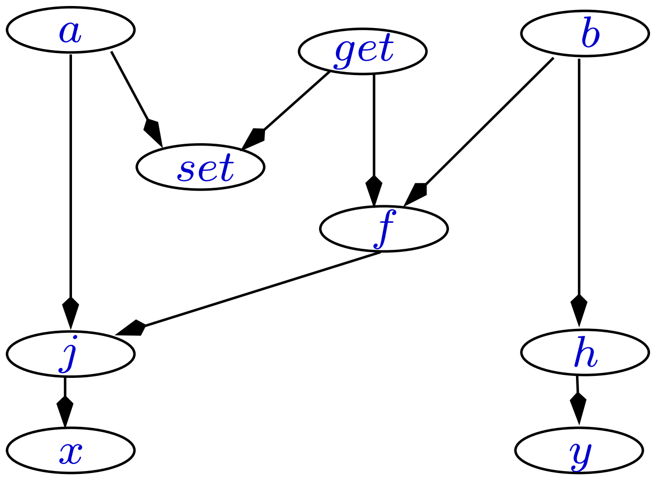
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a *(minimal) set of blocks* together with their inter-dependencies:

  this is called the *(Optimal) Static Scheduling Problem*

- only need to inline the *blocks dependency graph* within the caller



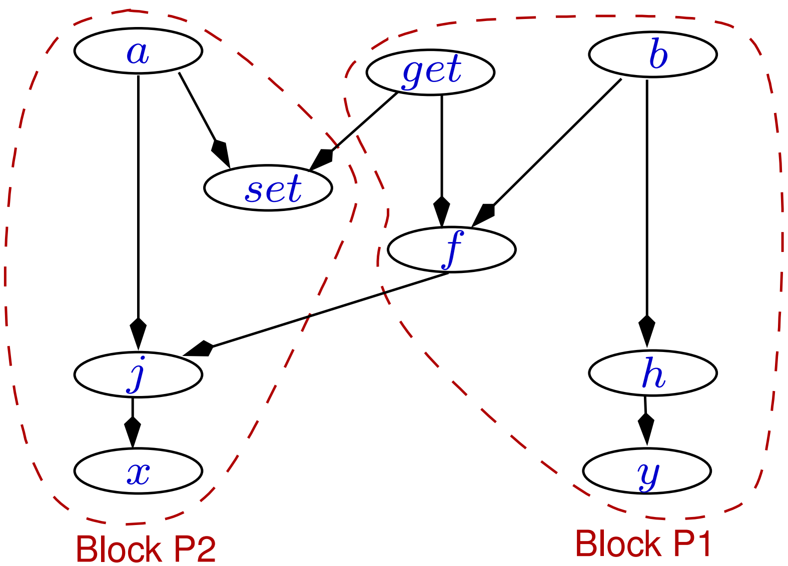
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a *(minimal) set of blocks* together with their inter-dependencies:

  this is called the *(Optimal) Static Scheduling Problem*

- only need to inline the *blocks dependency graph* within the caller



dependency analysis

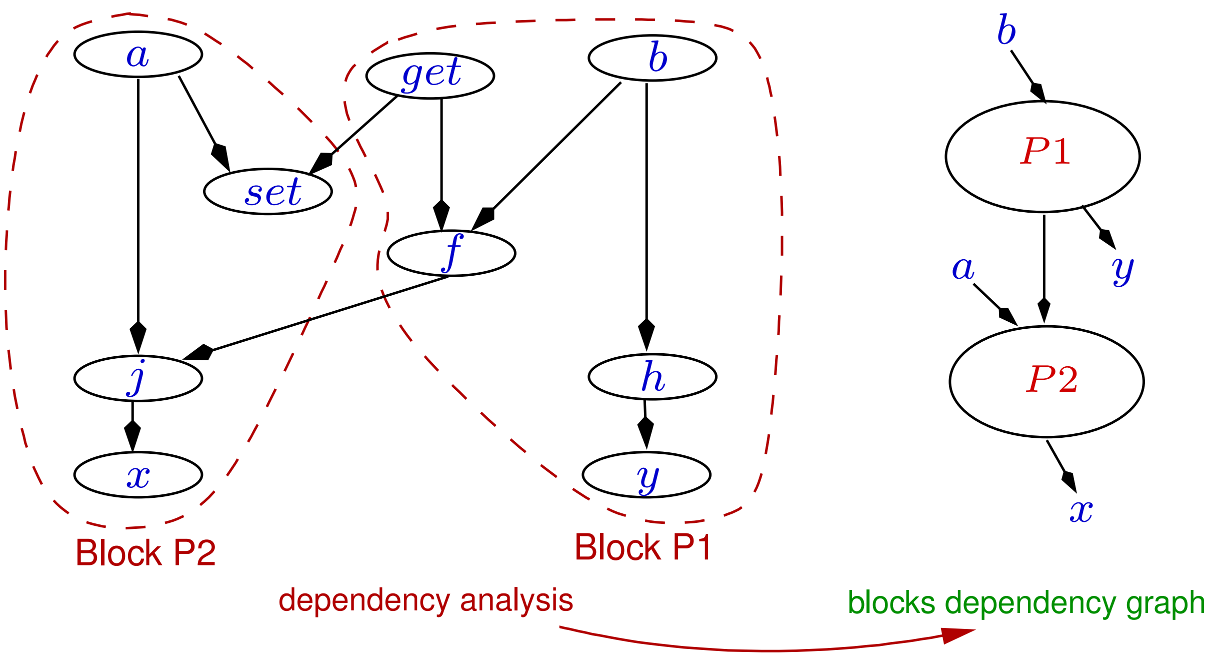
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a *(minimal) set of blocks* together with their inter-dependencies:

  this is called the *(Optimal) Static Scheduling Problem*

- only need to inline the *blocks dependency graph* within the caller



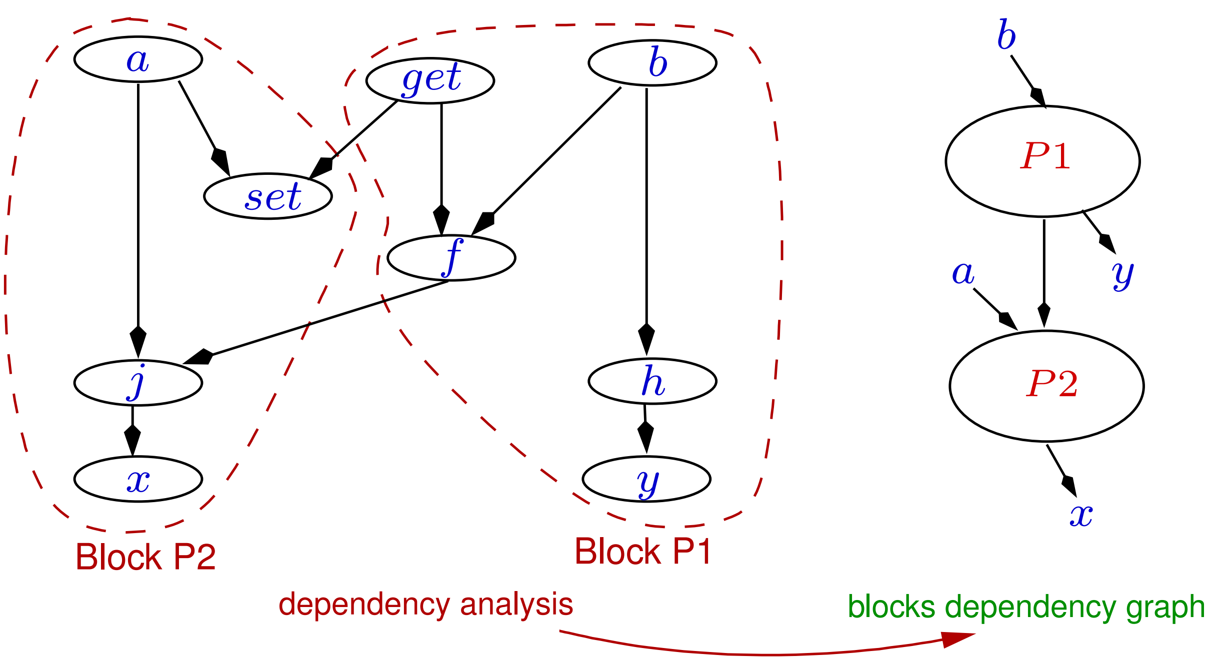
Grey-boxing

Some actions can be gathered without forbidding correct feedback loops:

- find such a *(minimal) set of blocks* together with their inter-dependencies:

  this is called the *(Optimal) Static Scheduling Problem*

- only need to inline the *blocks dependency graph* within the caller




```
proc P1 () {
    b ;
    get ;
    f ;
    h ;
    y ;
}
proc P2 () {
    a ;
    set ;
    j ;
    x ;
}
P1 before P2
```

+ sequential code

State of the Art

- Separate compilation of LUSTRE [Raymond, 1988]: *non optimal*
- Compilation/code distribution of SIGNAL [Benveniste *et al*, 90's]: *more general: conditional scheduling, not optimal*
- More recently, [Lublinerman, Szegedy and Tripakis, POPL'09]: *optimal, proof of NP-hardness, iterative search of the optimal solution through 3-SAT encoding*.

State of the Art

- Separate compilation of LUSTRE [Raymond, 1988]: *non optimal*
- Compilation/code distribution of SIGNAL [Benveniste *et al*, 90's]: *more general: conditional scheduling, not optimal*
- More recently, [Lublinerman, Szegedy and Tripakis, POPL'09]: *optimal, proof of NP-hardness, iterative search of the optimal solution through 3-SAT encoding*.

This work addresses the Optimal Static Scheduling Problem (OSS):

- proposes an encoding of the problem based on input/output analysis which gives:

  ↪ in (most) cases, an optimal solution in polynomial time

  ↪ or a 3-sat simplified encoding.
- practical experiments show that the 3-sat solving is almost never necessary

Formalization of the Problem

Definition: Abstract Data-flow Networks

A system $(A, I, O, \preceq)$*:*

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 precedence relation between actions.*

Definition: Compatibility

Two actions $x, y \in A$ *are said to be (static scheduling) compatible and this is written* $x \chi y$ *when the following holds:*

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

If two nodes are incompatible, gathering them into the same block creates an extra input/output dependency, and then forbids a possible feedback loop

Formalization of the goal

The goal is to find an *equivalence relation* (the set of blocks) implying compatibility plus a *dependence order* between blocks, that is, a *preorder relation*

Formalization of the goal

The goal is to find an *equivalence relation* (the set of blocks) implying compatibility plus a *dependence order* between blocks, that is, a *preorder relation*

Definition: (Optimal) Static Scheduling

A static scheduling over $(A, \preceq, I, O)$ *is a relation* $\precsim$ *satisfying:*

(SS-0) $\precsim$ *is a* pre-order *(reflexive, transitive)*

(SS-1) $x \preceq y \Rightarrow x \precsim y$

(SS-2) $\forall i \in I, \forall o \in O, \;\; i \precsim o \;\Leftrightarrow\; i \preceq o$

Formalization of the goal

The goal is to find an *equivalence relation* (the set of blocks) implying compatibility plus a *dependence order* between blocks, that is, a *preorder relation*

> ***Definition: (Optimal) Static Scheduling***
>
> *A static scheduling over* $(A, \preceq, I, O)$ *is a relation* $\precsim$ *satisfying:*
>
> *(SS-0)* $\precsim$ *is a* pre-order *(reflexive, transitive)*
>
> *(SS-1)* $x \preceq y \Rightarrow x \precsim y$
>
> *(SS-2)* $\forall i \in I, \forall o \in O, \;\; i \precsim o \;\Leftrightarrow\; i \preceq o$

Corrolary: let $\precsim$ be a S.S. and $(x \simeq y) \Leftrightarrow (x \precsim y \wedge y \precsim x)$ the associated equivalence, then $\simeq$ *implies* χ .

Formalization of the goal

The goal is to find an *equivalence relation* (the set of blocks) implying compatibility plus a *dependence order* between blocks, that is, a *preorder relation*

Definition: (Optimal) Static Scheduling

A static scheduling over $(A, \preceq, I, O)$ *is a relation* $\precsim$ *satisfying:*

- *(SS-0)* $\precsim$ *is a* pre-order *(reflexive, transitive)*
- *(SS-1)* $x \preceq y \Rightarrow x \precsim y$
- *(SS-2)* $\forall i \in I, \forall o \in O, \;\; i \precsim o \;\Leftrightarrow\; i \preceq o$

Corrolary: let $\precsim$ be a S.S. and $(x \simeq y) \Leftrightarrow (x \precsim y \wedge y \precsim x)$ the associated equivalence, then $\simeq$ *implies* χ .

Moreover, a Static Scheduling is optimal iff:

- *(SS-3)* $\simeq$ *has a minimal number of classes.*

Theoretical Complexity

- Lublinerman, Szegedy and Tripakis proved OSS to be NP-hard through a reduction to the *Minimal Clique Cover (MCC)* problem
- Since the OSS problem is an optimization problem whose associated decision problem is — *does it exist a solution with k classes?* —, they solve it iteratively by searching for a solution with $k = 1, 2, \ldots$ such as:

↪ for each k, encode the decision problem as a Boolean formula;

↪ solve it using a SAT solver

However, real programs do not reveal such complexity

- this complexity seems to happen for programs with a large number of inputs and outputs with complex and unusual dependences between them
- can we identify simple cases by analyzing input/output dependences?

The OSS problem encodes the **Minimal Clique Cover** (MCC) problem and is thus NP-hard, i.e., an OSS solver solves MCC.

Remark:

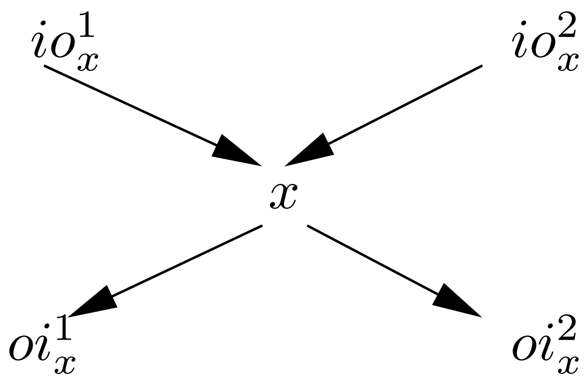
OSS is an **optimization problem** [see Garey and Johnson, 79] where the corresponding decision problem is: *does-it exist a solution with k classes?* Thus, a solution can be searched iteratively by trying $k = 1, 2, \ldots$.

Minimal clique cover (MCC), in terms of relations

let L be a a finite set, $\leftrightarrow$ a symmetric relation (i.e. given a non oriented graph), find a maximal equivalence relation $\simeq$ included in $\leftrightarrow$.

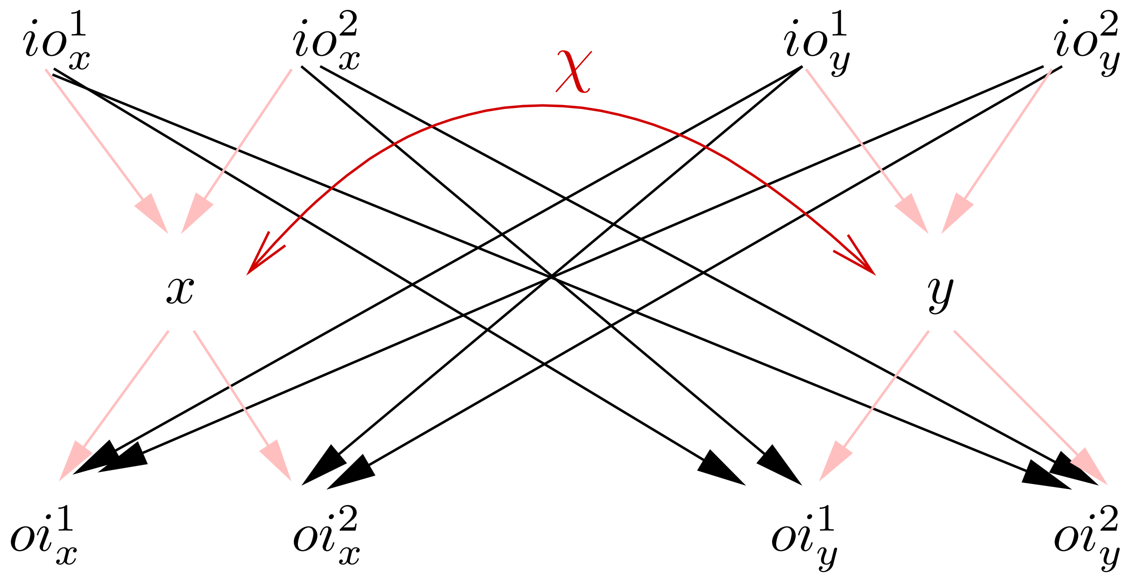
Let $(L, \leftrightarrow)$ be the data of a MCC problem, we build an instance $(A = L \uplus X, \preceq, I = X, O = X)$, by introducing a set of new input/output edges (X), and dependencies ($\preceq$) as follow:

- for each x in L, we have 4 extra variables io_x^1, io_x^2, oi_x^1 and oi_x^2, with the following dependencies:



- **N.B.** it enforces any extra variable to be incompatible with any other variable

- for each $x \leftrightarrow y$, we add 8 dependencies:



- **N.B.** it enforces local variables to be compatible iff $x \leftrightarrow y$)

We call this OSS instance the X-encoding of the MCC problem.

It is easily proven that:

- whatever be $\simeq$ an (optimal) solution of the MCC problem, then $\precsim \, = \, \simeq \cup \preceq$ is an optimal solution of X-encoded problem,
- whatever be $\precsim$ an (optimal) solution of the X-encoded problem, then the associated equivalence $\simeq$ is an optimal solution of the clique cover problem.

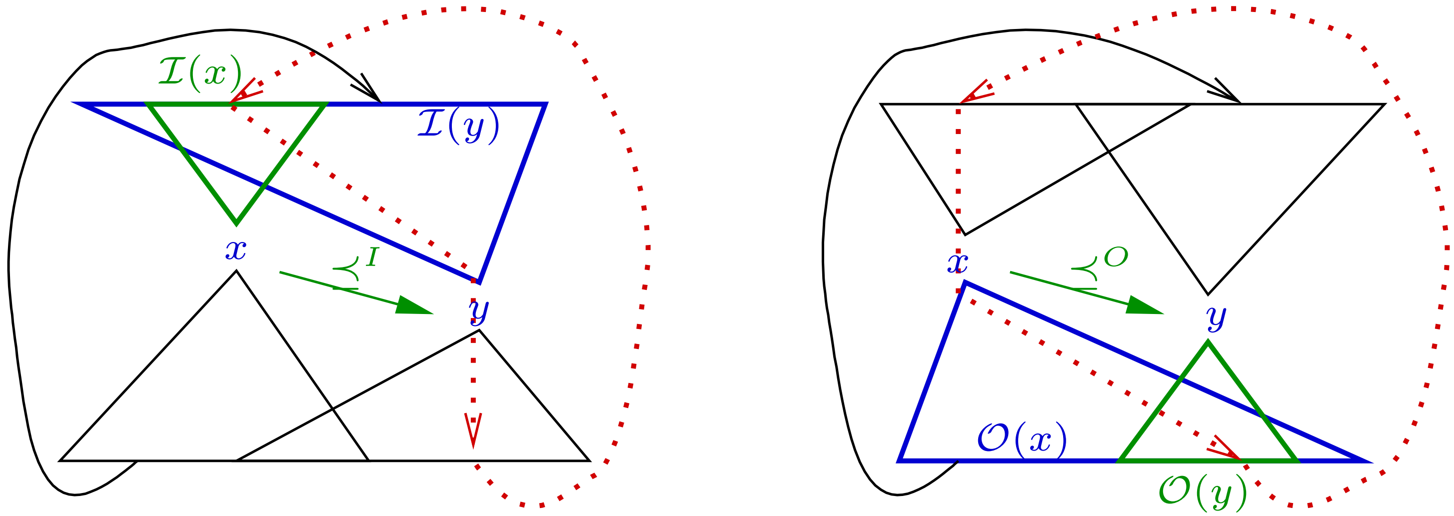
Conclusion:

- compatibility relations are **as general** as symmetric relations, thus NP-hardness.
- however, OSS instances that meet the general case have a large number of input and outputs with unusual input/output dependences

Analyse these input/output dependences to build a more efficient algorithm

Input (resp. output) pre-orders

Let $\mathcal{I}$ (resp. $\mathcal{O}$) be the input (resp. output) function:



It is *never the case* that x should be computed after y if either:

- $\mathcal{I}(x) \subseteq \mathcal{I}(y)$, noted $x \precsim^I y$, which is a valid of SS, (inclusion of inputs),
- $\mathcal{O}(y) \subseteq \mathcal{O}(x)$, noted $x \precsim^O y$, which is a valid SS. (reverse inclusion of outputs),

Input/output preorder

An even more precise preorder can be build by considering input preorder over output preorder:

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

N.B. a similar reasoning leads to the output/input preorder.

Properties

- $\precsim^{I_O}$ is a valid SS,
- moreover, it is *optimal for the inputs/outputs*:

$$\forall x, y \in I \cup \mathcal{O} \;\; x \simeq^{I_O} y \Leftrightarrow x \chi\, y$$

- it follows that, in any optimal solution, input/output that are compatible are necessarily in the same class (see proof in the paper)

Input-Set Encoding

- In any solution, the class of a node can be characterized by a subset of inputs or *key*: intuitivelly this key is the set of inputs that are computed before or with the node.
- As shown before, the only possible key for an input or output node x is $\mathcal{I}_{\mathcal{O}}(x)$

How to formalize what can be the key of an internal node?

- In any solution, the class of a node can be characterized by a subset of inputs or *key*: intuitivelly this key is the set of inputs that are computed before or with the node.
- As shown before, the only possible key for an input or output node x is $\mathcal{I}_{\mathcal{O}}(x)$

How to formalize what can be the key of an internal node?

Definition: KI-encoding

A KI-enc. is function $\mathcal{K} : A \mapsto 2^I$ which associate a key *to every node such that:*

(KI-1) $\forall x \in I \cup O; \mathcal{K}(x) = \mathcal{I}_{\mathcal{O}}(x)$

(KI-2) $\forall x, y \; x \preceq y \;\Rightarrow\; \mathcal{K}(x) \subseteq \mathcal{K}(y)$

Moreover:

(KI-opt) it is optimal if the image set is minimal.

Solving the KI-encoding

A system of (in)equations with a variable K_x for each $x \in A$:

- $K_x = \mathcal{I}_{\mathcal{O}}(x)$ for $x \in I \cup O$
- $\bigcup_{y \longrightarrow x} K_y \subseteq K_x \subseteq \bigcap_{x \longrightarrow z} K_z$ otherwise
  where $\rightarrow$ is the dependency graph relation (a concise representation of $\preceq$)

KI-encoding *vs* Static Scheduling

- a solution of KI ”is” a solution of SS (modulo key inclusion)
- any solution of SS *is not* a solution of KI (e.g, $\preceq$ itself, in general)
- but, *any optimal solution of SS* is also an optimal solution of KI (to the absurd, via Input/output preorder).

In other terms: the KI formulation is better than the SS one: it has less solutions, but does not miss any optimal one.

KI-encoding *vs* Static Scheduling

- a solution of KI "is" a solution of SS (modulo key inclusion)
- any solution of SS *is not* a solution of KI (e.g, $\preceq$ itself, in general)
- but, *any optimal solution of SS* is also an optimal solution of KI (to the absurd, via Input/output preorder).

In other terms: the KI formulation is better than the SS one: it has less solutions, but does not miss any optimal one.

Complexity of the encoding

- $O(n \cdot m^2 \cdot (\log m^2))$ where n is the number of actions, m the maximum number of input/outputs.
- That is, $O(n \cdot m \cdot B(m) \cdot A(m))$, where B is the cost of union/intersection between sets and A, the cost of insertion in a set.

Solving the KI-encoding: Example

$$K_a = \{a, b\} \quad K_b = \{b\} \quad K_x = \{a, b\} \quad K_y = \{b\}$$

$$\begin{aligned}
\emptyset \subseteq \quad & K_{\textit{get}} & \subseteq K_{\textit{set}} \cap K_f \\
K_a \cup K_{\textit{get}} \subseteq \quad & K_{\textit{set}} & \subseteq \{a, b\} \\
K_b \cup K_{\textit{get}} \subseteq \quad & K_f & \subseteq K_j \\
K_a \cup K_f \subseteq \quad & K_j & \subseteq K_x \\
K_b \subseteq \quad & K_h & \subseteq K_y
\end{aligned}$$

- The system to solve:
  - ↪ a variable K_x for each key
  - ↪ input/output keys are *mandatory*
  - ↪ set intervals for others

Solving the KI-encoding: Example

$$K_a = \{a,b\} \quad K_b = \{b\} \quad K_x = \{a,b\} \quad K_y = \{b\}$$

$$\begin{array}{rcl}
\emptyset \subseteq & K_{\mathit{get}} & \subseteq \{a,b\} \cap K_{\mathit{set}} \cap K_f \\
K_a \cup K_{\mathit{get}} \cup \{a,b\} \subseteq & K_{\mathit{set}} & \subseteq \{a,b\} \\
K_b \cup K_{\mathit{get}} \cup \{b\} \subseteq & K_f & \subseteq \{a,b\} \cap K_j \\
K_a \cup K_f \cup \{a,b\} \subseteq & K_j & \subseteq \{a,b\} \cap K_x \\
K_b \cup \{b\} \subseteq & K_h & \subseteq \{b\} \cap K_y
\end{array}$$

- Compute lower and upper bounds:

$\hookrightarrow$ $k_x^\perp = \bigcup_{y \to x} k_y^\perp$ and $k_x^\top = \bigcap_{x \to z} k_z^\top$

Solving the KI-encoding: Example

$$K_a = \{a,b\} \quad K_b = \{b\} \quad K_x = \{a,b\} \quad K_y = \{b\}$$

$$\emptyset \subseteq \quad K_{get} \quad \subseteq \{a,b\} \cap K_f$$

$$\{a,b\} \subseteq \quad K_{set} \quad \subseteq \{a,b\}$$

$$\{b\} \subseteq \quad K_f \quad \subseteq \{a,b\}$$

$$\{a,b\} \subseteq \quad K_j \quad \subseteq \{a,b\}$$

$$\{b\} \subseteq \quad K_h \quad \subseteq \{b\}$$

- Compute lower and upper bounds:

$\hookrightarrow$ $k_x^{\perp} = \bigcup_{y \longrightarrow x} k_y^{\perp}$ and $k_x^{\top} = \bigcap_{x \longrightarrow z} k_z^{\top}$

- Propagate, simplify: new equations, constant intervals, others

Solving the KI-encoding: Example

$$K_a = \{a,b\} \quad K_b = \{b\} \quad K_x = \{a,b\} \quad K_y = \{b\}$$

$$\begin{aligned} \emptyset &= K_{\textit{get}} \\ \{a,b\} &= K_{\textit{set}} \\ \{b\} &= K_f \\ \{a,b\} &= K_j \\ \{b\} &= K_h \end{aligned}$$

- Check for ”obvious” solutions:

$\hookrightarrow$ $\mathcal{K}^{\perp} : x \to k_x^{\perp}$

$\hookrightarrow$ strategy: compute as soon as possible

$\hookrightarrow$ not ”proven” optimal: $\emptyset$ not mandatory

Solving the KI-encoding: Example

$$K_a = \{a, b\} \quad K_b = \{b\} \quad K_x = \{a, b\} \quad K_y = \{b\}$$

$$\begin{aligned} K_{\mathit{get}} &= \{a, b\} \\ K_{\mathit{set}} &= \{a, b\} \\ K_f &= \{a, b\} \\ K_j &= \{a, b\} \\ K_h &= \{b\} \end{aligned}$$

- Check for ”obvious” solutions:

$\hookrightarrow$ $\mathcal{K}^\top : x \to k_x^\top$

$\hookrightarrow$ strategy: compute as late as possible

$\hookrightarrow$ *optimal:* all keys are mandatory

Dealing with complex systems

Let $\mathcal{S}$ be the simplified system, X be the set of actions whose key is still unknown, $\kappa_1, \cdots, \kappa_c$ be the c mandatory keys:

- try to find a solution with $c + 0$ classes:
  - $\hookrightarrow$ build the formula: $\mathcal{S} \bigwedge_{x \in X} \bigvee_{j=1}^{j=c} (K_x = \kappa_j)$
  - $\hookrightarrow$ call a SAT-solver...
- if it fails, try to find a solution with $c + 1$ classes:
  - $\hookrightarrow$ introduce a new variable B_1,
  - $\hookrightarrow$ build the formula: $\mathcal{S} \bigwedge_{x \in X} (\bigvee_{j=1}^{j=c} (K_x = \kappa_j) \vee (K_x = B_1))$
  - $\hookrightarrow$ call a SAT-solver...
- if it fails, try to find a solution with $c + 2$ classes, etc.

The prototype

- extract dependency informations from a LUSTRE (or SCADE) program
- build the simplified KI-encoded system (polynomial)
- check for obvious solutions (linear)
- if no obvious solution, iteratively call a Boolean solver.

We have considered three benchmarks made of the components comming from:

- the whole SCADE V4 standard library

  ↪ reusable programs, modular compilation is relevant

- two large industrial applications

  ↪ not reusable programs, less relevant

  ↪ but bigger programs, more likely to be *complex*

Results Overview

	# prgs	# nodes	# i/o	cpu	triv. (# blocks)	solved (# blocks)	other (# blocks)
SCADE lib.	223	av. 12	2 to 9	0.14s	65 (1)	158 (1 or 2)	
Airbus 1	27	av. 25	2 to 19	0.025s	8 (1)	19 (1 to 4)	
Airbus 2	125	av. 65 (up to 600)	2 to 26	0.2s	41 (1 to 3)	83 (1 to 4)	1^*

- as expected: programs in SCADE lib. are (small) and then simple
- but also in Airbus, even with "big" interface
- 1^*: not really "complex" (solved by a heuristic: intersection of $k_x^\top$)
- the whole test takes 0.35 seconds (CoreDuo 2.8Ghz, MacOS X); 350 LO(Caml).
- see source code in [**?**], Appendix.

Conclusion

- Optimal Static Scheduling is theoretically NP-hard
- thus it could be solved, through a suitable encoding, with a general purpose Sat-solver
- A polynomial analysis of inputs/outputs can give:
  - ↪ non trivial lower and upper bounds on the number of classes
  - ↪ a proved optimal solution in some cases
  - ↪ a optimized SAT-encoding that emphazises the sources of complexity
- Experiments show that complex instances are hard to find in real examples

References

*

References

[1] M. R. Garey and D. S. Johnson. *Computers and Intractability - A guide to the Theory of NP-completeness*. Freeman, New-York, 1979.

[2] 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.

[3] 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.

[4] Marc Pouzet and Pascal Raymond. Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation. *Journal of Design Automation for Embedded Systems*, 3(14):165–192, 2010. Special issue of selected papers from `http://esweek09.inrialpes.fr/`Embedded System Week. Extended version of [**?**].

[5] Pascal Raymond. Compilation séparée de programmes Lustre. Technical report, Projet SPECTRE, IMAG, July 1988.