Code distribution in the dSL environment for the synthesis of industrial process control

Bram De Wachter, Thierry Massart *
Brussels Free University, Computer Science Department
bdewacht@ulb.ac.be

December 7, 2004

Abstract

1 Introduction

Industrial process control goes hand in hand with distributed systems. This is due to the physically distributed nature of the environment that is being controlled through sensors and actuators. Development of such distributed reactive systems is a complicated task, even for an experienced programmer. The burden of combining the physical complexity of the process to control, the communications schemes of the distributed parts of the controlling system and the extreme reliability and robustness requirements make development of such systems very hard. This paper introduces the dSL concepts and environment, for the development of industrial process control. dSL eases this development by offering the programmer an event driven imperative language with OO concepts that has the benefit of automated distribution.

Transparent and optimized distribution are the key ideas contained in this paper. On a distributed system when a computation attempts to access data from another processor, communication has to be performed to satisfy the reference. Two mechanisms can be used to access remote data : data migration and computation migration [TvS02]. Several programming models use data migration, where the remote data is communicated to the demanding processor, or computation migration where the execution is migrated to the processor that has the data [HKW96], or any model that is some combination of both of them [WBC+92]. Our approach is mainly based on the use of computation migration and introduce a novel technique that enables transparent distribution and optimize the code with respect to some performance measure.

This paper is organized as follows : in the next section we introduce dSL , giving its simplified syntax and informal semantics that capture the most of dSL 's possibilities. Next, we discuss the choice of computation migration and its impact on the automated distribution of dSL . We explain the reason of performance deterioration implied by the distribution, and give a measure for it. Then we detail our technique that optimizes the code with respect to the performance measure. The last section gives the details on how we handle the execution of a distributed dSL program. Finally, conclusions end this paper.

2 The dSL concept

dSL is a supervision language associated with a full environment dedicated to control complex industrial processes. In this section we shortly present dSL 's predecessor the language SL and its environment and motivate dSL to ease the development of such control systems.

*Work supported by Bruxelles Capitales Region, grant no. RBC-BR 227/3298.

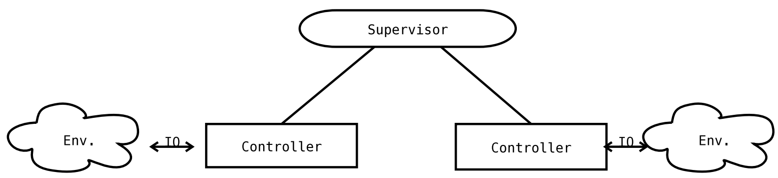


Figure 1: SL.

The language SL [1] ($_d$SL predecessor) is used to control and supervise industrial processes that are typically organized as shown in figure 1. Three entities can be distinguished 1. a supervisor, 2. the programmable controllers (called automata from here on) and 3. The environment. The environment concretizes the actual process that is controlled. It interacts with the automata that are basically made out of input/output hardware. The automata report the status of the environment to the supervisor that processes the input, and sends commands back to the automata and to the user to which it is interfaced. Since automaton and supervisor each have their role to play, their behavior is specified in a different way : automata are programmed in an assembler like language, while the supervisor's behavior is specified in SL. SL is an imperative language with limited OO concepts. It is event driven, where an event is specified by the change in some variable value. For instance, `when x > 0 then execMethod();` will trigger execMethod every time the variable x goes from a negative to a positive value.

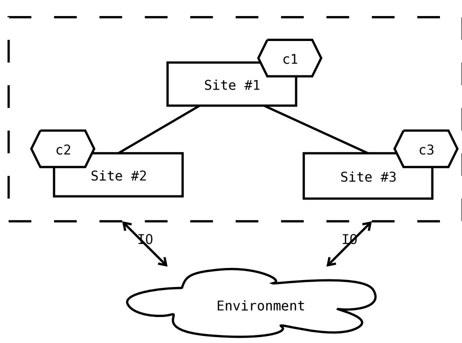


Figure 2: $_d$SL .

$_d$SL extends these features. Indeed, with the introduction of $_d$SL , the approach changes to figure 2. The essential idea is to unify the way automata and supervisor act to control the environment : we see the system as a set of sites with their respective capabilities (c_i). The main idea is to offer the programmer a centralized vision of the distributed application. A $_d$SL program is written as if the entire environment can be accessed without the need for explicit communication or synchronization primitives. The $_d$SL environment handles the code, and distributes automatically among the several sites based on their capabilities. These capabilities may describe for instance that a certain site has access to a certain actuator / sensor or that it may do floating point arithmetic. It is clear that this approach has many benefits such as 1. maintainability (only one language is used) 2. flexibility (changing an actuator / sensor from one site to another does not imply changes on the program), 3. simplicity (since communication/distribution is done implicitly, the programmer does not need to come up with synchronization schemes to handle particular tasks), ...

$_d$SL inherits the event-driven execution constructs under the form of `when` from its predecessor SL and is enriched with the capability for parallel execution through the `launch` construct which may be seen as an asynchronous method invocation. We added the capability to directly manage I/O variables declared with `input_byte` and `output_byte` as $_d$SL must be able to replace the assembler code that was running on the automata. In this paper, we will not detail any further the $_d$SL semantics, we just introduce a short example to concretize the main ideas behind $_d$SL .

[1] SL (Supervision Language) developed at Macq Electronique, Belgium

2.1 $_{\mathrm{d}}$SL example

To illustrate the $_{\mathrm{d}}$SL concepts, let us give a simplified example of a program which controls (reads) an input variable, linked for instance to a temperature sensor and switches on (off) the heater and an indicator in a control panel when the temperature is below 0 ^{o}C (above 20 ^{o}C).

```
global_var led, heater : output_byte; temp : input_byte; end_var

begin_method set_heater(state) (* Make the led's state correspond to
the heater's action *) heater := state; led := state; end_method

when temp < 0 then (* Turn on the heater *) set_heater (1); end_when

when temp > 20 then (* Turn off the heater *) set_heater (0); end_when

program (* Initially turn off heater and led indicator *) set_heater
(0); end_program
```

A $_{\mathrm{d}}$SL program is made up of 4 parts. 1. global variable declaration including all I/O variables 2. Method definitions 3. when instructions and 4 an initialization. Distribution of information must be added to the $_{\mathrm{d}}$SL program, for instance in our example, the temperature sensor and heater can be connected with the automaton A while the led indicator is connected with B.

No other distribution, communication or synchronization information is provided by the programmer.

The `WHEN` construct allows to formulate an event-handler; it is made up of two parts : 1. (*expr*) The condition that must change from `false` to `true` enabling the handler and 2. (*instruction_list*) the code that has to be executed in that case.

The `Launch` construct allows parallel execution. It may be seen as a asynchronous method invocation.

A simplified grammar of the $_{\mathrm{d}}$SL syntax can be found in Appendix A.

3 Distribution

Our goal is to transform a program that is written as if every I/O is localized on a single site into several distinct programs that communicate to each other using some sort of reliable communication channel. This *distribution* must be 1. correct i.e. the behavior of the centralized program must be the same as the behavior of the distributed program 2. reliable and 3. efficient.

As stated in the introduction, a variety of methods can be used to implement this distribution, ranging from data migration to computation migration. All of these methods consist of resolving the same basic problem : data objects and processing units are physically separated. Data and computation migration may be viewed as duals. Computation migration takes the advantage of spatial locality of the data objects while data migration takes advantage of their temporal locality.

Data migration can be used to describe any model that moves the data to the processing unit. Data migration goes hand in hand with data replication, a technique that generates several instances of the same data to improve performance but induces the cost of keeping the replicas consistent. With the $_{\mathrm{d}}$SL environment in mind, the use of data migration is impossible because of the difficulty to correctly implement an acceptable memory consistency model. A memory consistency model is absolutely necessary since it defines the set of allowable memory access orderings that could be observed on the set of sites participating in the distributed execution. Either the model is strict [Lam79], it allows only very few orderings, and simple to implement [IS99] but yields too bad performance because of the large amount of synchronization it induces. Or the memory consistency model is weakened [GLL^{+}98, BZ91, ISL96], it allows more executions and needs less synchronization, but its implementation [SFL^{+}94, CBZ91, KCZ92] is too complex to be implemented on our automata that are part of the $_{\mathrm{d}}$SL environment. Indeed, such models require complex algorithms that are too costly - in terms of communication, memory and processing resources - to be acceptable candidates for the controllers.

Computation migration, on the other hand, involves moving a thread of computation to the host or processor where the data is located. It is quite simple to implement on the $_{\rm d}$SL runtime system that uses a virtual machine common to automata and supervisor and is expected to have good results. Small messages (since only a part of the active thread's state is to be migrated) are communicated and less storage is needed (a complete copy of the program is not needed at all sites). Furthermore, the data model is single writer, which means that for a given data object only one site can modify its state, and this makes it far more easy to debug the system at real-time since data is not moved around. A last important motivation as to use computation migration is that actuators and sensors - that are physically attached to a certain site - are represented by data that is thus confined to that site.

Since our choice is to implement a distributed system using computation migration, the source program that was built upon the idea of a (fictive) centralized environment, needs to be distributed amongst different sites. We will see that our goal will consist in statically assigning each instruction to a single site. Finding such an assignment or instruction coloring which is efficient turns out to be quite difficult.

First of all, the coloring is not entirely free meaning that some instructions are constrained to a certain site. Consider for example an instruction that involves the assignment to an output variable : that instruction can clearly not be placed elsewhere than on the site that controls that particular output. We will refer to these constraints as the initial coloring.

Secondly, there is the performance penalty induced by the coloring. When two consecutive instructions are placed on different sites, the computation will be migrated at runtime from one site to another. The price to pay from the performance point of view is enormous, due to the difference between a processor's and network's speed. Therefore, our goal is to minimize these situations. We suppose that the execution of two consecutive instructions that are placed on the same site have no cost, while on different sites induce a costs of 1.

Remark *Two consecutive instructions with the same color have no performance penalty. Even if the data needed to execute the instructions is not localized on the executing site. This is caused by our hybrid distribution model : we assign a single color to each global variable (based on the capabilities of the sites they are defined/used on) and calculate the function Imposed so that the assignment to a global variable happens on the site that controls that global variable. Local variables may be (and are) migrated from one site to another. The local data is considered as internal or temporary data, so each definition of a local variable x (x is on the left-hand-side of an assignment) introduces a new x, which allows to apply any color to an instruction that defines the local variable x, cfr section 5 for more details.*

We express the performance penalty induced by the distribution by using an attribute grammar that calculates the mean number of migrations that will occur during the execution of a given program in function of the coloring. The resulting cost function can be obtained only when we suppose known the number of times each `WHILE` executes, and the probability of right and left branch of each `IF`. We leave this to the programmer as a tool to fine-tune the distribution that results from this cost-function.

The tree constructed by the grammar contains simple nodes (leaves) that represent basic instructions and composed nodes that represents the other kind of instructions. Simple nodes are $assign$, $call$, end_if, $launch$, $expr$ in $while$ and if. Composed nodes are $instr\_list$, if, $while$ and $instr$. Several attributes are common to simple nodes and composed nodes 1. $first$ is used to indicate the first (leftmost) simple node found in the tree derived from the node 2. $last$ is the last (rightmost) simple node, and 3. $cost$ expresses the performance penalty introduced by the node, i.e. the mean number of times the execution changes site in the nodes derived from the node. In the following, we suppose the possible colors are identified by an index which ranges from one to k and $\perp$ means no color (i.e. not colored yet). The attribute c that introduces the colors applies only on simple nodes. A coloring is thus a mapping from the c attributes of each element in the set of simple nodes to the set of colors representing the different sites. Finally, the attribute $imposed$ is the color imposed by the initial coloring, or $\perp$ if the simple node may take any color. From here on, we call the set of simple nodes the set P of instructions of the program, and an element i in that set is called an instruction.

We have then :

Definition 1 *The function Imposed expresses the initial coloring : it maps each instruction $i \in$ Instructions(P) to $\{\perp, 1, \cdots, k\}$ where $\perp$ means that i has no color, and $\{1, \cdots, k\}$ assigns a fixed color to*

i.

Cost function			
$instr\_list_0$	$\rightarrow$	$instr$ $instr\_list_1$	$instr\_list_0.cost :=$ $instr.cost + instr\_list_1.cost$ $+\Delta(instr.last, instr\_list_1.first)$ $instr\_list_0.first := instr.first$ $instr\_list_0.last := instr\_list_1.last$
$instr\_list$	$\rightarrow$	$instr$	$instr\_list.cost := instr.cost$ $instr\_list.first := instr.first$ $instr\_list.last := instr.last$
$instr$	$\rightarrow$	$\mathcal{I}$ $\mathcal{I} \in \{if, while, assign, call,$ $end\_if, launch\}$	$instr.cost := \mathcal{I}.cost$ $instr.first := \mathcal{I}.first$ $instr.last := \mathcal{I}.last$
if	$\rightarrow$	if $expr$ then $instr\_list_0$ else $instr\_list_1$ end_if	$if.cost := p \cdot [\Delta(expr.c, first(instr\_list_0))+$ $instr\_list_0.cost + \Delta(last(instr\_list_0, end\_if.c))]+$ $(1-p) \cdot [\Delta(expr.c, first(instr\_list_1))+$ $instr\_list_1.cost + \Delta(last(instr\_list_1, end\_if.c))]$ $if.first := expr$ $if.last := end\_if$
$while$	$\rightarrow$	while $expr$ then $instr\_list$	$while.cost := \omega[\Delta(expr.c, first(instr\_list))+$ $instr\_list.cost + \Delta(last(instr\_list), expr.c)]$ $while.first := expr$ $while.last := expr$
$assign$ (idem $call$, $end\_if, launch$)	$\rightarrow$	$id := expr$	$assign.cost := 0$ $assign.first := assign.last := assign$

With $Delta(i_1, i_2) \equiv 0$ if for coloring C, $C(i_1) = C(i_2)$, 1 otherwise.

Remark *The proposed function is an intra-procedural cost evaluation function. We therefore focus in this paper on one procedure at a time. A method call / launch is considered to have a single color. In the distribution, we will force entry and exit of each procedure to be of the same color.*

Remark *In if.cost, p* $\in]0, 1[$ *(we suppose p* $\neq 0$ *and p* $\neq 1$*), in while.cost,* $\omega \in \mathbb{N} \setminus \{1, 2\}$ *(we suppose that each while executes at least 2 times)* [2] *and may be different for each* `IF`*/*`WHILE`*. It expresses for an* `IF` *the probability that the 'true' branch is taken, and for a* `WHILE` *the number of times the loop is executed. This information must be introduced by the programmer.*

In the next section, we introduce a coloring technique that 1. assigns a color to each instruction 2. respects the initial coloring and 3. minimizes the performance penalty. The benefit of this technique is that it provides the $_d$SL concept with a transparent distribution : the initial coloring can be derived from the capabilities of each site and using that information, *coloring* distributes all other instructions.

4 Coloring

Definition 2 *A coloring C is a function that maps for each instruction i,* $i.c$ *to a color* $\in \{1, \cdots, k\}$.

Definition 3 *C is a correct coloring if for each instruction* i*, such that* $i.imposed \neq \perp$, $i.imposed = i.c$.

Definition 4 *The cost(P, C) for a given* $_d$*SL program P (=*$instr\_list$*) and a given coloring C is calculated by the attribute grammar defined in section 3, cost(P, C) = P.cost.*

[2] If these constraints are not respected, the results expressed in the following sections still hold. However, more irreducible graphs (see further on) are encountered.

Definition 5 *The optimal colorings $\mathfrak{C}^{opt}$ is the set of correct colorings C such that there is no correct coloring C' such that $cost(P, C') < cost(P, C)$*

Definition 6 *The coloring problem is the problem of finding for a given P and Imposed an element in the set $\mathfrak{C}^{opt}$.*

We use a graphical representation of the cost function under the form of a non directed graph. The graphical representation results from the simplification of the expression of $cost(P, C)$ for a given P and C to $cost(P, C) = \Sigma_i\, w_i \cdot \Delta(n_i, m_i)$. (The expression is reduced to a finite sum of basic terms of the form $w_i\Delta(n_i, m_i)$). Remark that a couple (n_i, m_i) is in a term of the simplified sum only if control may flow from instruction n_i to m_i (or from m_i to n_i). We obtain the weighted control flow graph(N, E, w) with nodes $N =$ set of instructions, edges $E : N \times N \equiv \cup_i\{n_i, m_i\}$ and weights $w : E \times \mathbb{R} \equiv (\{n_i, m_i\}, w_i)$. [3] Remark also that the edges are not directed, since $\Delta(i_1, i_2) = \Delta(i_2, i_1)$.

With this representation an assignment of a color to an instruction is the assignment of a color to a node in the graph that represents that instruction. The cost function that results from a coloring is then calculated as the sum of the weights of the edges that connect two nodes in the graph that have a different color. The graph is the graphical representation of the terms in $cost(P, C) = \Sigma_i\, w_i \cdot \Delta(n_i, m_i)$. We adopt this view to continue the discussion of the coloring problem. For a concrete example of the weighted control flow graph associated to a program, cfr figure 3.

Unfortunately, we have not found any polynomial solution to the coloring problem. Neither have we been able to prove that the problem is $\mathcal{NP}$-complete.

The problem for proving $\mathcal{NP}$-completeness of this problem is the fact that the control flow graph is not a general graph (and neither are the weights on the edges): unless some *goto* construct were introduced in the grammar in appendix A, only special cases can be constructed.

In the next section we show how - using a reduction technique - we can find either an optimal solution (by exhaustive search in a reasonable amount of time), or find satisfactory results using a local heuristics.

```
while C1 do (* executed k times *) if C2 then I1 (* probability p1 *)
else I2 end_if if C3 then I3 (* probability p2 *) end_if
end_while
```

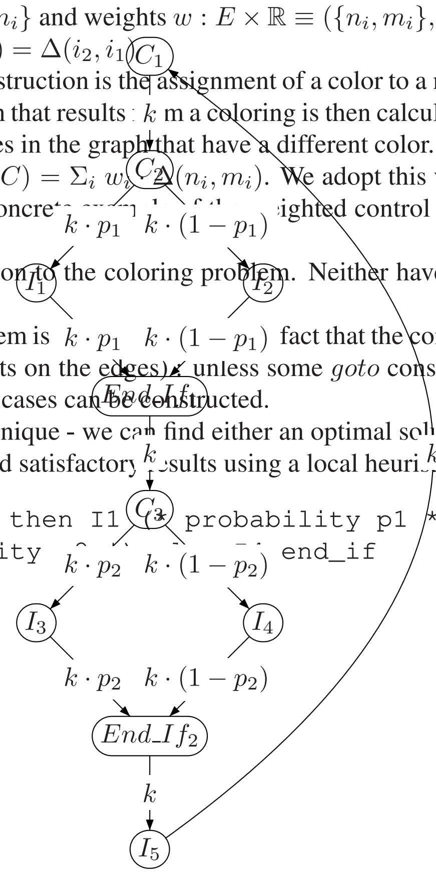


Figure 3: Example program.

4.1 The coloring problem is NP complete

To show that the above problem is NP complete, we use the results from [???] stating that the multiterminal cut for arbitrary (non planar) weighted graphs is NP complete, even with all weights set to 1. However, we still have to show that arbitrary graphs can be constructed from a syntactical correct dSL program.

We will therefore proceed by constructing a dSL program P for an arbitrary graph G, by induction on the size of G. P will contain only one sequence of instructions, thus showing that this problem is intractable even without the `while` or `if` constructs.

The main idea is to exploit the fact that an instruction i_1 is to be *merged* with an instruction i_2 if both are using (defining) the same global variable. Indeed, since a each global variable must be distributed on

[3] The graph obtained here is not exactly a control flow graph as defined in [ASU86], since nodes in this graph are not basic blocks. Each basic block of n instructions in [ASU86] is split up in our representation into a chain of n nodes with 1 instruction.

a given site, two instructions using the same variable must therefore be localised on the same site. An example of a simple $_{\text{d}}$SL sequence with such an additional contraint is given in figure 4.

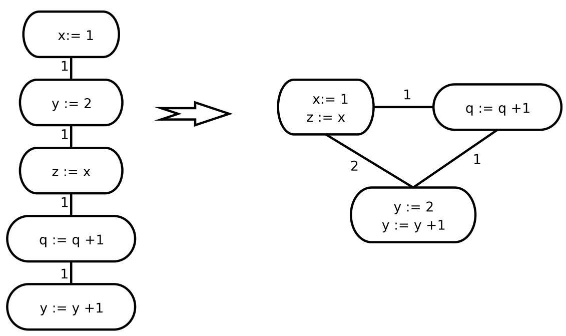


Figure 4: An example of a merge in the coloring graph

Lemma 1 *For any graph G, there exists a syntactical correct $_{\text{d}}$SL program P such that the coloring graph of P equals G.*

Let $G = (V, E)$ with $|V| = n$ and let $L : V \mapsto 2^{instr(P)}$ be a labeling function that records the mapping between the nodes in G and the corresponding instructions in P.

We will prove this lemma by induction on the size of G. By induction hypothesis, let P be a syntactical correct program containing a single sequence of assignements, and such that $f(P) = G$. We will show that we can obtain $G' = (V', E')$, where $V' = V \cup \{v\} \wedge E' = E \cup \{\{v, v_1\}, \{v, v_2\}, ..., \{v, v_k\}\}$ where $\{v_1, v_2, ..., v_k\} \subseteq V$ by adding some assignments in P.

$\forall j \in [1..k]$: let i be an instruction such that $i \in L(v_j)$, since P is a list of instructions, we have a predecessor (p_i) and a successor (s_i) instruction for i. Remark that the case where i is the first or last instruction in P is easy to handle by duplicating i (in that case, $p_i = i$ or $s_i = i$). We change P in such a way that $p_i; i; s_i$ becomes $p_i; i; i_v; i; s_i$;, where i_v is a fresh instruction that contains the variable x_v.

It is easy to see that the above construction is such that

- *the resulting program still is a sequence of instructions*
- *exactly one new node (corresponding to i_v) is inserted (after the merging of all instructions using x_v)*
- *the previously existing edges remain the same*
- *the flow of control between i_v and i inserts the edges in $E' \setminus E$.*

■

Theorem 1 *The coloring problem is NP Complete.*

Lemma 1, shows how to construct a $_{\text{d}}$SL program with an arbitrary coloring graph. In [???], ??? prove that the resulting Colored Multiterminal Cut problem is NP Complete. ■

4.2 2-Way local reduction

Because we did not find a polynomial solution to the coloring problem, and have proven that the problem is in $\mathcal{NP}$, we searched for a way to reduce the size of the problem to instances that are acceptable to be treated exhaustively. The reduction method presented here reduces the size of the initial problem by finding classes of nodes that may have the same color. We prove that there is an element in $\mathfrak{C}^{\mathtt{opt}}$ that assigns the same color to each set of nodes that make up these classes. The reduction is based upon the following observation :

> If the graph contains an uncolored node n_1 with an edge e to n_2, let $e_1 \cdots e_k$ be the edges from n_1 with $e_1 = e$. If $\sum_{i=2\cdots k} w(e_i) \leq w(e_1)$ holds, then n_1 and n_2 can be assigned the same color.

Using this condition, we can reduce the graph by 1 node, merging n_1 and n_2, repeating this reduction until no edge remains that can be reduced, results in the reduced graph. Note that this reduction method allows to reduce a sequence of simple instructions (of the form $I_1; I_2; I_3; I_4$), and for compound statements it can take care of the `IF`'s condition and `END_IF` nodes (consider I_1; IF C THEN I_2; END_IF; I_3).

Proof of the safety of the reduction (by contradiction) :
(Let $e_i = \{n_1, m_i\}$ and $w(e_i) = w_i$)
$\exists C \in \mathfrak{C}^{\mathtt{opt}} : C(n_1) = C(n_2)$
Suppose there is no such C, then $\forall C \in \mathfrak{C}^{\mathtt{opt}}$:
$\Delta(C(n_1), C(n_2)) = 1$
let $C'(n) = \begin{cases} C(n) & n \neq n_1 \\ C(n_2) & n = n_1 \end{cases}$
then $cost(P, C) - cost(P, C') =$
$w_1.\Delta(C(n_1), C(n_2)) + \sum_{i=2\cdots k} w_i.\Delta(C(n_1), C(m_i))$
$-w_1.\Delta(C'(n_1), C(n_2)) - \sum_{i=2\cdots k} w_i.\Delta(C'(n_1), C(m_i))$
$=$
$w_1 + \sum_{i=2\cdots k} w_i.\Delta(C(n_1), C(m_i))$
$- \sum_{i=2\cdots k} w_i.\Delta(C'(n_1), C(m_i))$
$\geq$
$w_1 - \sum_{i=2\cdots k} w_i$
$\geq$
0 (by hypothesis that $\sum_{i=2\cdots k} w(e_i) \leq w(e_1)$ holds)

$\Rightarrow cost(P, C) \geq cost(P, C')$ which is in contradiction with the fact that $C \in \mathfrak{C}^{opt}$, and $C' \notin \mathfrak{C}^{opt}$.

4.3 3-Way Local reduction

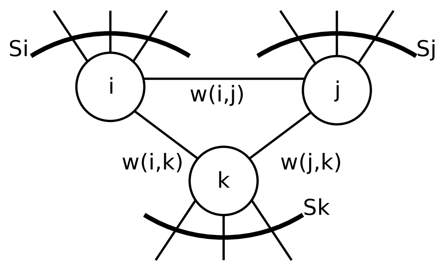


Figure 5: Global reduction theorem

A generalization of the 2-way local reduction can be stated as follows :

Theorem 2 *Given a weighted undirected graph G(V,E,w), a set of terminals $\mathcal{T}$ and three different vertices $n_1, n_2, n_3 \in (V \setminus \mathcal{T})$, if the following conditions hold*

$$2 \cdot (w(n_1, n_2) + w(n_1, n_3)) > w(n_1)$$
$$2 \cdot (w(n_2, n_3) + w(n_1, n_2)) > w(n_2)$$
$$2 \cdot (w(n_3, n_1) + w(n_3, n_2)) > w(n_3)$$
$$3 \cdot (w(n_1, n_2) + w(n_1, n_3)) + w(n_3, n_2)) > w(n_1) + w(n_2) + w(n_3) - max(S_{n_1}, S_{n_2}, S_{n_3})$$

where

$$w(n_1) = \sum_{\{(n_1,n)\in E\}} w(n_1, n)$$
$$S_{n_1} = w(n_1) - w(n_1, n_2) - w(n_1, n_3)$$
$$S_{n_2} = w(n_2) - w(n_1, n_2) - w(n_2, n_3)$$
$$S_{n_3} = w(n_3) - w(n_1, n_3) - w(n_2, n_3)$$

then for any k-multiterminal cut of G (w.r.t $\mathcal{T}$) there exists a k-multiterminal cut of minimal weight $V_1, ..., V_k$: $\exists p : \{n_1, n_2, n_3\} \subseteq V_p$ (i.e. n_1, n_2, n_3 are in the same partition) ■

4.4 Global reduction

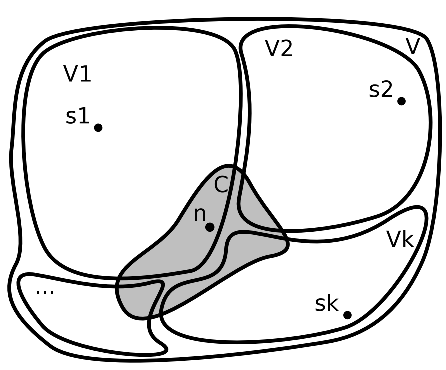


Figure 6: Global reduction theorem

Theorem 3 *Let $G(V, E, w)$ be an undirected weighted graph ($E \subseteq V \times V$, $w : E \mapsto \mathbb{Z}^+$). Let $V \supset \mathcal{T} = \{s_1, ...s_k\}$. Let $(C, V \setminus C)$ be an st-cut between $n \in V \setminus \mathcal{T}$ and t^* ($n \in C, t^* \notin C$) of minimal weight, where t^* is obtained by merging all nodes in $\mathcal{T}$.*

There exists a minimal k-cut of G ($V_1, V_2, ..., V_k$) separating all terminals in $\mathcal{T}$ ($s_i \in V_i, V_i \cap V_{j \neq i} = \phi, \cup_j V_j = V$) such that $\exists l \in [1..k] : \forall n' \in C : n' \in V_l$. ■

Proof:
Suppose a minimal k-cut C^* and, whithout any loss of generality, $l = 1 (n \in V_1)$. We define a partition of V in $V_1', ..., V_k'$ as follows :

$$V_1' = V_1 \cup C \tag{4.1}$$

$$\begin{aligned} V_{j \neq 1}' &= V_j \setminus (V_j \cap C) \\ &= V_j \setminus C \end{aligned} \tag{4.2}$$

We show that $V_1', ...V_k'$ is a k-multicut, noted $C^{*'}$, with weight less than (or equal to) the weight of C^*, proving the existence of a k-multicut described in the theorem.

$C^{*'}$ is a k-multicut, since it is easy to verify that

- $V_i' \cap V_{j \neq i}' = \phi$
- $\cup_i V_i' = V$
- $\forall i : s_i \in V_i$
- $\forall i : s_i$ is isolated from $s_{j \neq i}$

Let us show that $C^{*'}$ has weight less than (or equal to) the weight of C^*. We note $w(A, B) = \sum_{\{(x,y)|x \in A, y \in B\}} w(x, y)$. Furthermore, let $w(X) = w(X, \overline{X})$, where $\overline{X} = V \setminus X$.
Using these notations, we have the following rules :

$$w(X, Y) = w(Y, X) \tag{4.3}$$

$$w(X, Y \cup Z) = w(X, Y) + w(X, Z) - w(x, Y \cap Z) \tag{4.4}$$

$$w(X, Y \setminus Z) = w(X, Y) - w(X, Y \cap Z) \tag{4.5}$$

Note also that since C^* is a k-cut :

$$V_i \cap V_{j \neq i} = \phi \tag{4.6}$$

The weights of the two k-cuts $C^{*'}$ and C^* can be expressed in terms of their partitions (resp. V_i' and V_i) as follows :

$$w(C^{*'}) = \sum_{j \neq 1} w(V_1', V_j') + \sum_{i \neq 1, j > i} w(V_i', V_j') \tag{4.7}$$

$$w(C^*) = \sum_{j \neq 1} w(V_1, V_j) + \sum_{i \neq 1, j > i} w(V_i, V_j) \tag{4.8}$$

In addition, we use two st-cuts for proving $w(C^{*'}) \le w(C^*)$:

$$
\begin{aligned}
(4.9)\qquad w(C) &= w(C, V \setminus C) \\
&= w(C, (\cup_j V_j) \setminus C) \\
&= \sum_j w(C, V_j \setminus C) && \text{by (4.4), (4.6)} \\
(4.10)\qquad &= w(C, V_1 \setminus C) + \sum_{j \neq 1} w(C, V_j \setminus C) \\
(4.11)\qquad w(C \cap V_1) &= w(C \cap V_1, V \setminus (C \cap V_1)) \\
&= w(C \cap V_1, (\cup_j V_j) \setminus (C \cap V_1) \\
&= \sum_j w(C \cap V_1, V_j \setminus (C \cap V_1)) && \text{by (4.4), (4.6)} \\
(4.12)\qquad &= w(C \cap V_1, V_1 \setminus C) + \sum_{j \neq 1} w(C \cap V_1, V_j)
\end{aligned}
$$

In order to calculate $w(C^{*'}) - w(C^*)$, we will express all terms of $C^{*'}$ in terms of C^* :

$$
\begin{aligned}
w(V_1', V_j') &= w(V_1 \cup C, V_j \setminus C) && \text{by (4.1), (4.2)} \\
&= w(V_j \setminus C, V_1) + w(V_j \setminus C, C) \\
&\quad - w(V_j \setminus C, V_1 \cap C) && \text{by (4.3), (4.4)} \\
&= w(V_1, V_j) - w(V_1, V_j \cap C) + w(V_j \setminus C, C) \\
(4.13)\qquad &\quad - w(V_1 \cap C, V_j) + w(V_1 \cap C, V_j \cap C) && \text{by (4.3), (4.5)} \\
w(V_{i \neq 1}', V_{j > i}') &= w(V_i \setminus C, V_j \setminus C) && \text{by (4.2)} \\
&= w(V_i \setminus C, V_j) - w(V_i \setminus C, V_j \cap C) && \text{by (4.5)} \\
&= w(V_j, V_1) - w(V_j, V_i \cap C) \\
(4.14)\qquad &\quad - w(V_j \cap C, V_i) + w(V_j \cap C, V_i \cap C) && \text{by (4.3), (4.5)}
\end{aligned}
$$

Using equations (4.13) and (4.14) in (4.7), we can express the difference between $C^{*'}$ and C^* using (4.10) and (4.12) :

$$
\begin{aligned}
& w(C^{*'}) - w(C^*) = \\
(4.15)\qquad & \quad w(C) - w(C \cap V_1) \\
(4.16)\qquad & \quad + \sum_{j \neq 1} (-w(V_1, V_j \cap C) + w(V_1 \cap C, V_j \cap C)) \\
(4.17)\qquad & \quad + \sum_{i \neq 1, j > i} (-w(V_j, V_i \cap C) - w(V_j \cap C, V_u) + w(V_i \cap C, V_j \cap C)) \\
(4.18)\qquad & \quad - w(V_1 \setminus C, C) + w(V_1 \cap C, V_1 \setminus C)
\end{aligned}
$$

Showing that $w(C^{*'}) - w(C^*) \le 0$ since

- (4.15) ≤ 0 because C is a minimal cut (remark that $C \cap V_1$ is an s-t cut between n and t^*)
- (4.16) ≤ 0 because $(V_1 \cap C) \subseteq V_1$
- (4.17) ≤ 0 because $(V_i \cap C) \subseteq V_i$ and $w(X, Y) \ge 0$
- (4.18) ≤ 0 because $(V_1 \cap C) \subseteq C$

which proofs the theorem.

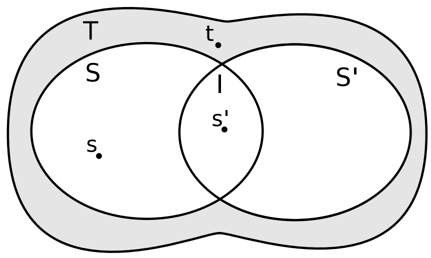


Figure 7: Max-min-ST theorem

Theorem 4 *Given a weighted graph $G(V, E, w)$ and three different vertices $s, s', t \in V$. We define the set max-min-ST(s,t) as the set of nodes left connected to s by the minimum cut in G of maximal cardinality separating s and t.*

For any three vertices s, s', t of V, we have the following property : if $s' \in$ max-min-ST(s,t), then max-min-ST$(s',t) \subseteq$ max-min-ST(s,t). ■

Proof : By contradiction. Let S =max-min-ST(s,t), S' =max-min-ST(s',t) and $S' \not\subseteq S$. Furthermore, let $I = S' \cap S'$ and $T = V \setminus (S \cup S')$. We have:

$$\begin{aligned} w(I) &= w(I, V \setminus I) \\ &= w(I, (S' \setminus I) \cup (S \setminus I) \cup T) \\ &= w(I, S' \setminus I) + w(I, S \setminus I) + w(I, T) && \text{by (4.4)} \\ w(S') &= w(S', S \setminus I) + w(S', T) \\ &= w(S', S \setminus I) + w((S' \setminus I) \cup I, T) \\ &= w(S', S \setminus I) + w(S' \setminus I, T) + w(I, T) && \text{by (4.4)} \end{aligned}$$

and therefore

$$\begin{aligned} & w(I) - w(S') = \\ \text{(4.19)} \quad & w(I, S \setminus I) - w(S', S \setminus I) \\ \text{(4.20)} \quad & + w(I, S' \setminus I) - w(S' \setminus I, T) \end{aligned}$$

Since $I \subseteq S'$ (4.19) is less than 0. It remains to show that (4.20) is also less than 0 to show the contradiction. Since

$$\begin{aligned} w(S) &= w(S, V \setminus S) \\ &= w(S, T \cup (S' \setminus I)) \\ \text{(4.21)} \quad &= w(S, T) + w(S, S' \setminus I) && \text{by (4.4)} \\ w(S \cup S') &= w(S \cup S', T) \\ \text{(4.22)} \quad &= w(S, T) + w(S', T) - w(I, T) && \text{by (4.4)} \end{aligned}$$

Hence

$$\begin{aligned} w(S \cup S') - w(S) &= w(S', T) - w(I, T) - w(S, S' \setminus I) && \text{by (4.22)} \\ \text{(4.23)} \quad &> 0 && \text{by Hypothesis} \end{aligned}$$

and

$$\begin{aligned} w(S, S' \setminus I) &< w(S', T) - w(I, T) && \text{by (4.23)} \\ \text{(4.24)} \quad &< w(S' \setminus I, T) && \text{by (4.5)} \end{aligned}$$

since

$$w(I, S' \setminus I) \leq w(S, S' \setminus I) \qquad \text{by (4.24), } I \subseteq S \tag{4.25}$$

The contradiction follows from (4.20), (4.25) and $S \subseteq T$.

Theorem 5 *Given a weighted graph $G(V,E,w)$ and three different verticies $s, s', t \in V$. Let S =max-min-ST(s,t), and S' =max-min-ST(s',t). If $I = S \cap S' \neq \phi \wedge S \neq I \wedge S' \neq I$, then $w(I, S \setminus I) = w(I, S' \setminus I)$.* ■

Proof by contradiction. Suppose $w(I, S \setminus I) > w(I, S' \setminus I)$. We have

$$\begin{aligned} w(S' \setminus I) = & \, w(S') - w(I, V \setminus S') \\ & + w(I, S' \setminus I) \end{aligned}$$

To show the contradiction, since $S' =$ max-min-ST(s',t) and $S' \setminus I$ is an st-cut(s',t), it is sufficient to show that

$$w(I, S' \setminus I) < w(I, V \setminus S')$$

but

$$\begin{aligned} w(I, V \setminus S') &> w(I, S \setminus S') && (S \setminus S') \subset (V \setminus S') \\ &= w(I, S \setminus I) \end{aligned}$$

and

$$w(I, S \setminus I) > w(I, S' \setminus I) \qquad \text{by Hypothesis}$$

Theorem 6 *Given a weighted graph $G = (V,E,w)$ and two vertices $s, t \in V$. We have $|$max-min-ST$(s,t)| = 1$. I.e. there is only one maximal size minimal ST cut.*

■

Proof by contradiction. Suppose we have $V_s, V'_s \in$ max-min-ST(s,t) such that $V_s \neq V'_s$. Let $I = V_s \cap V'_s, T = V \setminus (V_s \cup V'_s)$, it is easy to see that $V_s \neq I \wedge V'_s \neq I$, else V_s (resp. V'_s) would not be a maximal size cut. Figure 8 shows V_s and V'_s. We have :

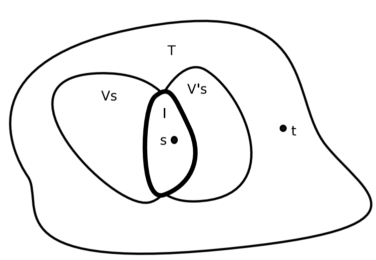


Figure 8: V_s and V'_s

$$\begin{aligned} w(I) &\geq w(V_s) \\ w(V_s \setminus I, I) + w(V'_s \setminus I, I) + w(I,T) &\geq w(V_s, V'_s \setminus I) + w(V_s, T) \\ w(V_s \setminus I, I) + w(V'_s \setminus I, I) + w(I,T) &\geq w(V_s, V'_s \setminus I) + w(V_s \setminus I, T) + w(I,T) \quad \text{as } V_s = (V_s \setminus I) \cup I \\ w(V_s \setminus I, I) + w(V'_s \setminus I, I) &\geq w(V_s, V'_s \setminus I) + w(V_s \setminus I, T) \end{aligned}$$

as $I \subseteq V_s$, we have that

$$w(V_s' \setminus I, I) = w(I, V_s' \setminus I) \leq w(V_s, V_s' \setminus I)$$

therefore, we must have

$$w(V_s \setminus I, I) \geq w(V_s \setminus I, T) \tag{4.26}$$

Now, let's compute $w(V_s \cup V_s')$, note that $V_s \cup V_s'$ is also a cut-ST(s,t).

$$\begin{aligned} w(V_s \cup V_s') &= w(V_s \setminus I, T) + w(V_s' \setminus I, T) + w(I, T) \\ &\leq w(V_s \setminus I, I) + \underbrace{w(V_s' \setminus I, T) + w(I, T)}_{=w(V_s', T)} && \text{by 4.26} \\ &\leq w(V_s \setminus I, I) + w(V_s', T) \\ &\leq w(V_s \setminus I, V_s') + w(V_s', T) && \text{as } I \subseteq V_s' \\ &\leq w(V_s') \end{aligned}$$

Thus, $V_s \cup V_s'$ is a minimal ST-cut(s,t). But we have $|V_s \cup V_s'| > |V_s'|$, thus $V_s, V_s' \notin$ max-min-ST(s,t) and we have a contradiction.

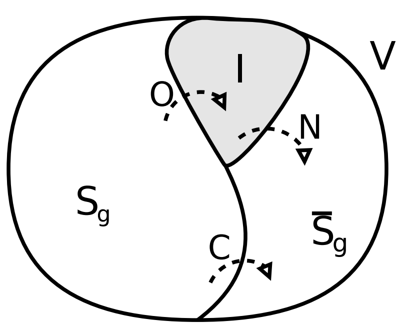


Figure 9: Theorem 7

Theorem 7 *The algorithm presented in [GT88] calculating the maximum flow in $O(nm\log(\frac{n^2}{m}))$-time also calculates max-min-ST* ■

Proof : by contradiction
In [GT88] the authors prove that it is possible to calculate a minimal cut $(S_g, \overline{S}_g)$ with $s \in S_g \wedge t \in \overline{S}_g$ in $O(nm\log(\frac{n^2}{m}))$-time. We will use their notation to prove that the min-cut calculated by their algorithm is in fact the unique minimal cut of maximal size.
The authors use $g(v,w) : E \mapsto \mathbb{R}^+$ as the preflow function (here we may suppose that the algorithm terminated and that the preflow is a legal flow). G_g is used to indicate the residual graph and $c(v,w) : E \mapsto \mathbb{R}^*$ indicates the capacities of the edges in E. In addition, $(S_g, \overline{S}_g)$ is defined as the partition of V such that $\overline{S}_g$ contains all vertices from which t is reachable in G_g and $S_g = V \setminus \overline{S}_g$.
We will use the following lemma from [GT88] :

Lemma 2 *When the first stage terminates, $(S_g, \overline{S}_g)$ is a cut such that every pair v, w with $v \in S_g$ and $w \in \overline{S}_g$ satisfies $g(v,w) = c(v,w)$* ■

Suppose that there exists another cut $(C', \overline{C'})$ such that $|C'| > |S_g|$.
Remark that $S_g \subseteq C'$ because of theorem 4 and $s \in C' \cap S_g$.
Let $I = C' \cap \overline{S}_g$. Note that $I \neq \phi$ since $|C'| > |S_g|$. We split the boundaries between S_g, I and $\overline{S}_g$ in three sets : O (old boundary) , N (new boundary), and C (common boundary).

- $O \subseteq E = (v, w) : v \in S_g \setminus I \wedge w \in I$
- $N \subseteq E = (v, w) : v \in I \wedge w \in \overline{S}_g \setminus I$
- $C \subseteq E = (v, w) : v \in S_g \setminus I \wedge w \in \overline{S}_g \setminus I$

By definition of $(S_g, \overline{S}_g)$, we know that

$$g(O) = c(O) \tag{4.27}$$

We also know that since $(C', \overline{C'})$ and $(S_g, \overline{S}_g)$ are both minimal cuts :

$$\begin{aligned} w(O) + w(C) &= w(N) + w(C) \\ w(O) &= w(N) \end{aligned} \tag{4.28}$$

Remark that since g is a legal flow, the flow entering I must be equal to the flow getting out of I, which means that

$$g(O) = g(N) \tag{4.29}$$

Combining (4.27),(4.28) and (4.29) leads to a contradiction: since the edges in N are saturated, t is not reachable from any $n \in I$ in G_g which means that $I = \phi$.

4.5 Coloring the reduced graph

We tested this reduction method on a large set of randomly generated graphs (conform to the grammar in appendix A). The first conclusion we made from our test cases is that all of them are reduced to graphs that have very few uncolored nodes. For that reason, the reduction method yields very satisfying results for practical cases, since an exhaustive search can be performed.

			Irreducible				
Nodes	colors	colored %	Total %	2-4	5-7	8-10	¿ 10
18	2	8	2	100	0	0	0
32	2	8	4	100	0	0	0
53	2	8	10	98	1.5	0.5	0
89	2	8	25	93	6.8	0.2	0
148	2	8	45	79	19	1.4	0.6
248	2	8	67	60	30	7.8	2.2
18	2	20	3	100	0	0	0
30	2	20	9	99	1	0	0
52	2	20	21	97	3	0	0
88	2	20	40	87	11	1.5	0.5
145	2	20	63	70	25	4.3	0.7
246	2	20	80	46	32	16	6
18	4	20	3	100	0	0	0
31	4	20	9	99	1	0	0
52	4	20	22	97	3	0	0
88	4	20	41	87	12	1	0
148	4	20	62	71	25	4	0
246	4	20	80	45	33	15	7
18	2	40	3	100	0	0	0
30	2	40	9	100	0	0	0
52	2	40	20	98	2	0	0
89	2	40	35	91	8	1	0
148	2	40	53	78	19	2.2	0.8
246	2	40	70	58	29	10	3

Figure 10: Results

More interesting is that for a big majority of moderate sized graphs the reduction method ends in a configuration where the coloring problem becomes straightforward. In figure 10, some results from random generated graphs are summarized. Each row represents results that are calculated on a set of 5000 random generated graphs, with the number of nodes in the first column followed by the number of different imposed colors and the amount of nodes that have one of the imposed colors. The fourth column shows the number of irreducible graphs, and the following columns detail for these graphs the spreading of how many uncolored nodes are - after reduction - connected to uncolored nodes. For example 100% in the column *2-4* on the first line means that all irreducible graphs (2% of the total of 5000 generated graphs) had 2,3 or 4 uncolored nodes that where connected to uncolored nodes.

The results in figure 10 show clearly that the number of different colors has no effect on the efficiency of the reduction. Indeed, our reduction method makes only the difference between colored and uncolored nodes.

At first sight, one could think that the number of nodes in figure 10 is insufficient to represent real programs. This is not the case because of two reasons. First of all, we proposed an intra-procedural evaluation function, so the number of nodes is the number of instructions in one procedure at a time. Secondly, no use of sequential simple instructions is made in our test cases, since sequential simple instructions are always reducible. This means that the number of nodes are the number of composed instructions (i.e. `IF`, `WHILE`) used in one procedure : only automatically generated code could approach the amount of 250 nodes in one procedure.

We can see that in most of the cases, the reduced graph has uncolored nodes that are only connected to colored nodes. In that case, we may find $C \in \mathfrak{C}^{\mathtt{opt}}$ immediately by evaluating for each n (with $e_j = \{n, m_j\}$ the edges connected to n) : $min_c\{\sum_{e_j} w(e_j) | color(m_j) \neq c\}$.

In some cases, uncolored nodes are connected to uncolored nodes, and no efficient solution is found yet. Either we apply an exhaustive search - note that even for very large graphs this is still applicable if the number of colors that are involved is not too high - or a local heuristic. For a complete example (optimal reduction vs. non-optimal reduction), see appendix B.

4.6 A local heuristic

Transforming the graph $G(N, E, w)$ by merging two arbitrary nodes n_1 and n_2 (where at least one is not colored) moves the cost away from the optimal cost. The difference between the optimal cost before reduction and after reduction can be majorated using $E_1 = \{\{n_1, n\} \in E | n \neq n_2\}$ and $E_2 = \{\{n_2, n\} \in E | n \neq n_1\}$. These are respectively the set of edges that connect n_1 to a node different from n_2, and the set of edges that connect n_2 to a node different from n_1.

The difference between the optimal cost before and after reduction is then majorated by $max\{0, \sum_{e \in E_1} w(e) + \sum_{e \in E_2} w(e) - w(n_1, n_2)\}$.

Proof of the majoration theorem :
Let $cost_1$ be the optimal cost before reduction, $cost_2$ after reduction. For simplicity, instead of removing a node from the graph $G(N, E, w)$, we look at the reduction as if it only forces the coloring to assign the same color to the nodes n_1 and n_2.
1. If there is a coloring that has a cost of $cost_1$ before reduction, where n_1 and n_2 have the same color, then the reduction has no effect on the cost, and $cost_1 = cost_2$.
2. Suppose before reduction, $\nexists C \in \mathfrak{C}^{\mathtt{opt}} : C(n_1) = C(n_2)$. Then, $\forall C \in \mathfrak{C}^{\mathtt{opt}}$ before reduction :
Let C_r be the coloring after reduction that assigns to each node the same color as the optimal coloring before reduction, except for the nodes n_1 and n_2. $\forall n \in N \setminus \{n_1, n_2\} : C_r(n) = C(n)$ with $C_r(n_1) = C_r(n_2)$. Let $cost_r$ be the cost of C_r.
$cost_2 - cost_1 \leq cost_r - cost_1$ (since $cost_2$ is optimal).
$cost_r - cost_1 \leq \sum_{e \in E_1} w(e) + \sum_{e \in E_2} w(e) - w(n_1, n_2)$. (construction of C_r).

1. and 2. $\Rightarrow cost_2 - cost_1 \leq max\{0, \sum_{e \in E_1} w(e) + \sum_{e \in E_2} w(e) - w(n_1, n_2)\}$.

A local heuristic consists of finding the edge that minimizes the majoration of the non-optimal reduction, and reduces the graph using that edge. The obtained graph may then be reduced using the optimal reduction method until it finds a solution, or needs another non-optimal reduction.

Some results of the efficiency of this heuristic can be found in figure 11. The results are obtained on randomly generated graphs based on the grammar in appendix A. Only irreducible graphs where retained, and an exhaustive search was performed to test the local heuristic. The first column gives the number of uncolored nodes that are connected to other uncolored nodes after reduction. These are the nodes for which the heuristic is needed. The next column gives the number of graphs that where encountered, and the next column shows the mean number of times the heuristic (and reduction) had to be applied to be able to color the entire graph. The last column measures the efficiency of the local heuristic by calculating the cost obtained by applying the heuristic divided by the optimal cost that is obtained by exhaustive search.

5 Distributed execution

In this section we explain how *distributed execution* is handled : how the execution may change site when two consecutive instructions are assigned a different color. We refer to this event as a *site-hop*, and explain how we use *def-use chains* - a classical data flow analysis [ASU86] - to build the right messages that will be communicated over a reliable channel to assure correct execution.

Nodes	Cases	Reductions	Heur/Exh
2	2111	1	1.026
3	1133	1.760	1.031
4	1468	2.262	1.031
5	1302	2.971	1.037
6	1236	3.591	1.042
7	1046	4.234	1.059
8	893	4.956	1.058
9	785	5.672	1.039
10	720	6.301	1.058
11	585	6.988	1.052
12	473	7.841	1.046
13	425	8.329	1.049
14	304	9.174	1.060
15	276	10.09	1.040
16	215	10.25	1.039
17	168	11.22	1.065
18	116	11.81	1.044
19	96	12.61	1.034
20	71	13.32	1.056
21	56	14.78	1.060
22	43	14.81	1.062
23	38	15.73	1.075
24	22	17.13	1.046

Figure 11: Local Heuristic : Results

As shown in figure 12, a site-hop combines activation of some instructions on a distant site with value passing of variables that are needed to continue correct execution on that site. There may be additional transmissions, if some future site needs a value that changed on the current site : in figure 12, x is transmitted from site A to C before the active site changes from A to B.

Definition 7 *A control flow graph is a directed graph(Nodes, Edge), where $Nodes$ is the finite set of nodes and $Edge$ a relation between the nodes. $n_1\ Edge\ n_2$ means that control can flow from n_1 to n_2. Each node n has two sets associated to it : the set use is the set of all used variables, the set def is the set of all defined variables.*

Definition 8 *$Succ(n)$ is the set of immediate successors of a given node, this is the set $\{n' \in Nodes | n\ Edge\ n'\}$.*

Definition 9 *A variable is defined when its value is written.*

Definition 10 *A variable is used when its value is read.*

Definition 11 *A colored control flow graph is a control flow graph(Node,Edge,C,color) where C is a finite set of colors and $color$ is a function $Node \mapsto C$ that assigns a color ($\in C$) to each node.*

Definition 12 *A node n in the control flow graph may be of any of these forms : 1. an assignment $x := f(y_1, \cdots, y_k)$, 2. a condition $f(y_1, \cdots, y_k)$ and 3. the empty instruction $skip$. In the first case $use(n) = \{y_1, \cdots, y_k\}$ and def$(n) = \{x\}$, in the second case $use(n) = \{y_1, \cdots, y_n\}$ and def$(n) = \phi$, and the third case $use(n) =$ def$(n) = \phi$.*

Definition 13 *The nodes that are accessible from a node n in the control flow graph is noted by $Path(n) = \{n_1, \cdots, n_k \in Nodes | \forall i, 1 < i < k : n_i \in succ(n_{i-1})\}$*

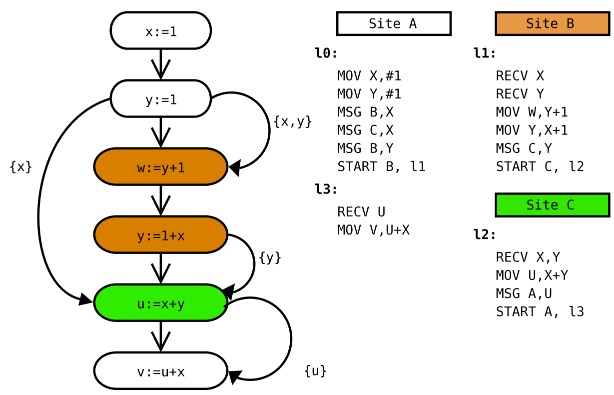


Figure 12: Value passing.

To construct these messages, we need a complete knowledge of what variables are used and defined, and more important in some cases, where (on what paths in the control flow graph) these definitions and uses take place. For each definition of a variable, we need to know whether or not the defined variable can be used before it is redefined, and if it is the case, we need to know on which site(s) the variable may be used. Using this information, we could send the new value at once, but to increase performance, we choose to postpone transmission until a site-hop occurs.

The aim of our message-construction algorithm is to enrich the colored control flow graph with two functions : 1. $T : Nodes \mapsto \{(x, c) | x \in Var, c \in C$: the value of variable x must be sent to the site associated to color $c\}$ and 2. $R : Nodes \mapsto \{(x, n) | x \in Var, n \in Nodes$: node n might send the value of variable $x\}$. The algorithm ensures that the functions T and R enable $_{\mathrm{d}}$SL 's distributed execution, meaning that all communications are present in these functions and that only the supplementary *start request* messages are needed to ensure correct distributed execution.

To introduce the algorithm, we give some definitions used in data-flow analysis.

Definition 14 *A variable is live at a given node in the control flow graph if there exists a path in the control flow graph starting at that node, on which the variable is used before it is defined.* $Live(n) = \{x \in Var | \exists n_1, \cdots, n_k \in Nodes : n_1 = n \wedge \forall i = 2 \cdots k : n_i \in succ(n_{i-1}) \wedge x \in use(n_k) \wedge \forall i = 1 \cdots k-1 : x \notin def(n_i)\}$

Definition 15 *We say that a node n' in the control flow graph uses a definition of x at node n if n defines x and n' uses x and if there exists a path in the control flow graph from n to n' along which x is not redefined.* [4]

Definition 16 *The def-use function $DU(n, x)$ gives the set of all nodes n' at which x is used and if there exists a path from n to n' along which x is not redefined.* [5]

All variables that are live are sent over to the sites where the variables are first used, except for those variables where the destination and source site are the same. Formally, the set of variables to send to the destination site $\mathcal{S} \subset Var \times Site$ can be defined as $\{(x, s) \in Var \times Site | \exists n_1, \cdots, n_k \in Nodes : \forall i, 1 < i \leq k : n_i \in succ(n_{i-1}) \wedge x \in use(n_k) \wedge \forall i, 1 < i < k : x \notin \text{def}(n_i) \wedge s = color(n_k) \wedge color(n_k) \neq color(n_1)\}$.

Remark *This definition may cause a variable to be sent to multiple destinations sites, as it is the case for variable x from A to B and C (see figure 12). This is preferred from the situation where x is first sent to B and then from B to C, since this is considered less robust.*

[4] As show in figure 13, node n' can use definitions of x from various nodes : the execution will determine which definition will be effectively used.

[5] Again, the figure 13 shows that $DU(n, x)$ gives the set of nodes which may use the definition of x provided by n.

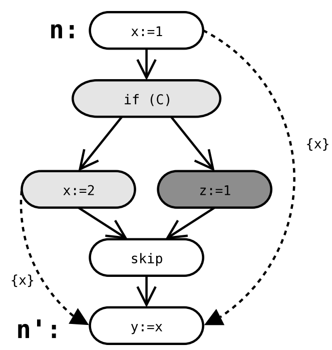


Figure 13: Multiple receptions.

We use *definition-use chains (du-chains)* ([ASU86]) to calculate which variables to send, and where to send them. To explain how we exploit this classic data flow information, consider a colored control flow graph.

Definition 17 *A monochrome-set of a node n is the maximal set of nodes reachable from n, in the colored control flow graph, having the same color. The set of such nodes is defined by* $P(n) = \{n' \in Node | \exists n_1, \cdots n_k : \forall i = 2 \cdots k : n_i \in succ(n_{i-1}) \wedge \forall i = 1 \cdots k : color(n_i) = color(n) \wedge n_1 = n'\}$

Definition 18 *The exit boundaries of a monochrome-set are the nodes of that set that have at least one successor that is not in the monochrome-set.* $P_x(n) = \{n' \in P(n) | \exists n'' \in succ(n'), color(n') \neq color(n'')\}$, $P_{xn}(n) = \{(n_1, n_2) | n_1 \in P_b(n) \wedge n_2 \in succ(n_1) \wedge color(n_1) \neq color(n_2)\}$.

Definition 19 *The entry boundaries of a monochrome-set are the nodes of that set that have at least one predecessor that is not in the monochrome-set.* $P_e(n) = \{n' \in P(n) | \exists n'' \in Node, n' \in succ(n'') \wedge color(n'') \neq color(n')\}$.

To explain how the message constructing algorithm works (cfr figure 15), consider a node n in the colored control flow graph that defines a variable `x` (for example `x := y + z`). We can use the function $DU(n, x)$ that calculates all the nodes that use this definition of x. For each $n' \in DU(n, x)$ with a different color than n, we know that we have to send the value of x. The total number of messages to send caused by this instruction is the number of colors of the nodes in $DU(n, x)$ that have a color other than $color(n)$.

Two strategies are possible, we can either send the value of x at once to the sites that are associated to the color of the instructions in $DU(n, x)$, or we can postpone the message until we actually leave the current site. We choose the more efficient second solution, and show where the transmission is made. The benefit of this approach is that of sparing bandwidth on the used communication channels and processing power for the composition and emission of the messages. Imagine for example a loop that changes `x` constantly while remaining on the same site. Clearly, sending a message each time `x` changes will introduce a severe performance penalty.

Clearly, the transmission is to be done in one of the boundary nodes of the monochrome-set started at n. Indeed, the nodes in $P_{xn}(n)$ are exactly those nodes that leave the current site, but not all of those nodes may lead to a use of x. In fact, because we postponed the transmission, we could end up in a situation where the calculated $du - chain$ does not hold any more. This is the case for $(n_1, n_2) \in P_{xn}(n)$ where x is not live at n_2. An example of this case is presented in the right branch on fig 14. Note that for each given destination site (the color of an instruction in $DU(n, x)$ different than $color(n)$) at least one boundary node exists where the transmission must be made (if not, the set $DU(n, x)$ was erroneous).

Since we want to associate the set of couples (x, c) of variables to transmit to certain sites, we slightly modify the colored control flow graph to avoid situations in which one node in the boundary of a color-path has (at least) two outgoing edges where the transmissions are not the same over these edges (cfg figure 14).

The transformation of the colored control flow graph is done by inserting dummy nodes (skip instruction, where $use = \text{def} = \phi$) as follows : for each couple of nodes $(n_1, n_2) \in P_{xn}(n)$ such that

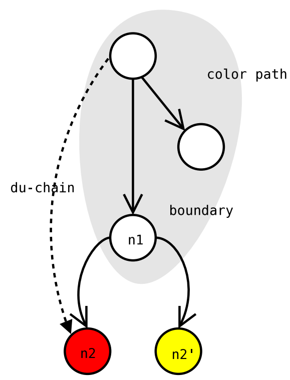


Figure 14: Postponing transmissions.

$\#[succ(n_1)] > 1$ remove the edge $n_1\ Edge\ n_2$, insert a dummy node n_d, and insert edges $n_1\ Edge\ n_d$ and $n_d\ Edge\ n_2$, state with $color(n_d) = color(n_1)$. This replaces n_1 in $P_x(n)$ by n_d (and thus (n_1, n_2) in P_{xn}) by (n_d, n_2)).

The construction of the reception set R, is straightforward : we know that for each node $n' \in DU(n, x)$ with different color than $color(n)$ we build a message that is sent to $color(n')$. We have to identify the nodes on site $color(n')$ that receive this message, so we can add the message into their R set. $P_e(n')$, is a safe approximation of that set since it includes all initial nodes through which control may flow activating the site with $color(n')$. It is also an approximation since there may be nodes in $P_e(n')$ that are not in any path in the control flow graph starting from n'. We filter these nodes out.

With the two sets, T and R, calculated, it is easy to insert the right code for the back end compiler. For each $n|(x, c) \in T(n)$ emit the instruction `send x to c from n`, for each $n|(x, m) \in R(n)$ emit the instruction `if from m then receive x`. Not only the value of `x` is communicated, but each variable must have a unique identifier that is sent along.

Message-constructing algorithm :
Input :   a colored control flow graph Output : $T : Nodes \mapsto \{(x, c) \mid x \in Var, c \in C : x$ is needed at the site with color $c\}$ $R : Nodes \mapsto \{(x, n) \mid x \in Var, n \in Nodes : n$ might send $x\}$.
$\forall n, \text{def}(n) \neq \phi, \forall x \in \text{def}(n)$ **do** $\forall n_1 \in P_x(n)$ **do** $\forall n_d \in DU(n_1, x)$ with $color(n_d) \neq color(n_1)$ **do** $T(n_1) = T(n_1) \cup \{(x, color(n_d))\}$ $\forall n_2 \in P_e(n_1)$ with $n_2 \in Path(n_1)$ **do** $R(n_2) = R(n_2) \cup \{(x, n_1)\}$ **od**   **od**   **od** **od**

Figure 15: Message construction.

Remark *The use of the variable's identifiers can be omitted if a total order on the identifiers is defined and variables are sent / received with respect to that order.*

Remark *It is important that the order in which messages are sent is guaranteed, since the start-request sent when a site hop occurs must be made after the reception of the messages that carry the state of variables. Therefore, the implementation may include a counter n_c for each $c \in C$. Each time the site with color c is activated, it checks if the number of received messages is equal to its counter n_c, then it resets n_c to 0. When a message is sent to a site with a certain color c, n_c is incremented. The counters n_c are passed along during each site-hop, they are the run-time verifiers of the number of sent and received messages.*

As shown above, the value passing part of the distributed execution of $_{\text{d}}$SL can be calculated by the help of the functions $DU(n, x)$, $Path(n)$, $P_x(n)$ and $P_e(n)$. The first two functions can easily be implemented using the algorithm described in [ASU86]. For functions $P_x(n)$ and $P_e(n)$, we can use a depth first search on the control flow graph to define the color-paths (observe that $n_2 \in succ(n_1) \wedge color(n_1) = color(n_2) \Rightarrow P_{xn}(n_2) \subseteq P_{xn}(n_1)$).

6 Conclusions

We presented $_{\text{d}}$SL , a language for the design and synthesis of industrial process control. We showed that $_{\text{d}}$SL eases the task of developing such applications by offering a virtual centralized environment to the programmer that is transparently distributed into a physical distributed configuration. We use computation migration for the distribution's model, because of the limited resources on the automata that are part of the system. Using this model, we identified the sources of performance deterioration (i.e. the *site-hops* and gave a precise measure for it). This measure, the expected number of migrations during execution, is finally used to show that our distribution algorithm has optimal performance.

Although we could not prove that the problem of finding the optimal distribution is in $\mathcal{NP}$, we showed that -using a safe reduction algorithm- the optimal solution can be found in practice by performing an exhaustive search on the reduced problem. As an alternative to the exhaustive search, we also introduced a local heuristic, that is based upon the minimization of the majoration of the penalty induced by reducing the graph using an arbitrary edge.

In the last section, we showed how the distributed execution of $_{\text{d}}$SL is handled, and introduced a message construction algorithm that is based upon *def-use* chains.

7 Future work

For the coloring problem, the irreducible graphs must be studied and might lead to some clue to find either a polynomial optimal solution, or prove its $\mathcal{NP}$-completeness. Therefore, other characteristics than the fact that for a given node no edge has a weight that is greater than the weights of all other edges connected to that node must be found that describe an irreducible graph.

As for the executed distribution, complex types are part of the $_{\text{d}}$SL 's complete grammar (such as structures, pointers and OO-programming). The dynamical nature of these concepts make it hard to build all communications messages at compile time. A possible solution is to use *continuation procedures* (described in [HKW96]), which makes each function dependent on its calling context, and use an inter-procedural pointer analysis to be able to make static decisions on the use of pointers.

Detailed semantics must be designed for the complete $_{\text{d}}$SL language, which may open the way to specification and verification of $_{\text{d}}$SL programs, a real must in the world of industrial process control. Another necessity is to proof that the distribution is *correct*, meaning that the transparent distributed program is conform with the semantics of the original program.

Finally, an implementation is on its way that defines -amongst a parser, compiler and distributer- a virtual machine unifying the different platforms (i.e. controller and supervisor). We are hoping to release a virtual machine that shows the advantages of $_{\text{d}}$SL on accessible platforms like the $Lego\ Mindstorms^{tm}$. This implementation may also be extended and finally, a case study is worked out.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., Reading, Mass., 1986.

[BDWS02] T. Massart B. De Wachter and S. Saeednia. Efficient heuristics for the optimal task labeling problem, submitted. 2002.

[BZ91] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Pittsburgh, PA (USA), 1991.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.

[GLL+98] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, October 1988. ISSN:0004-5411.

[HKW96] Wilson C. Hsieh, M. Frans Kaashoek, and William E. Weihl. Dynamic computation migration in DSM systems. 1996.

[IS99] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[ISL96] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, 1996.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[Lam79] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers, 28(9):690–691*, sept. 1979.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 297–306. ACM Press, 1994.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems, principles and paradigms*. Prentice Hall, 2002.

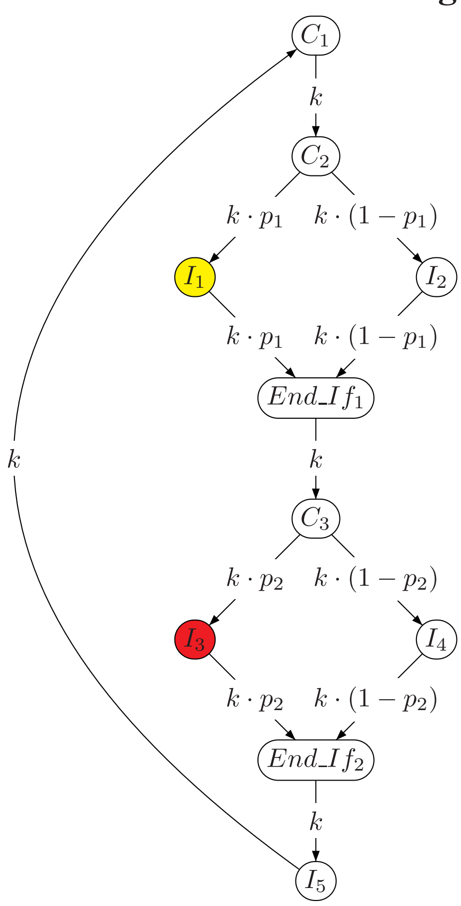
[WBC+92] William E. Weihl, Eric A. Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson C. Hsieh, Anthony D. Joseph, Carl A. Waldspurger, and Paul Wang. PRELUDE: A system for portable parallel software. In *Parallel Architectures and Languages Europe*, pages 971–973, 1992.

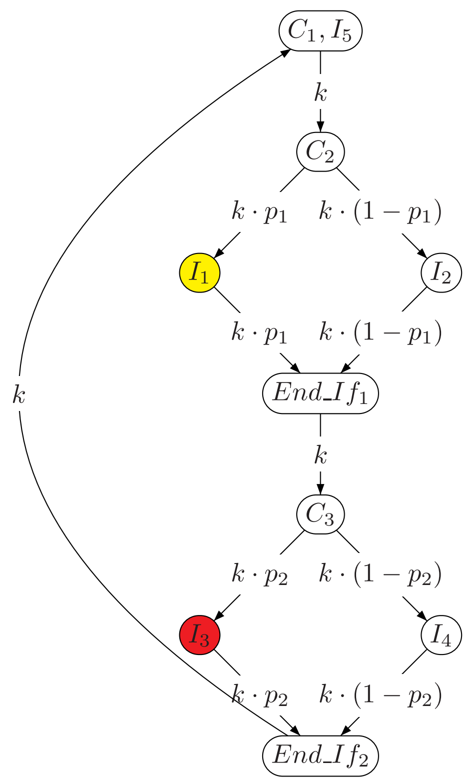
A Grammar

Remark *This grammar is a reduced version of the complete dSL grammar, and to keep it compact, some rules may introduce ambiguity.*

Dsl_Program	⟶	*global_var_declaration methods events init*
methods	⟶	*method_declaration methods* — ϵ
events	⟶	*when events* — ϵ
init	⟶	"**program**" "**begin_var**" *id_list* "**end_var**" *instruction_list* "**end_program**"
global_-var_declaration	⟶	"**global_var**" *var_list* "**end_var**"
var_list	⟶	*id_list* "**:**" *var_type* — *id_list* "**:**" *var_type* "**;**" *var_list*
id_list	⟶	*id* — *id* "**,**" *id_list*
var_type	⟶	"**int**", "**input_byte**", "**output_byte**"
method_declaration	⟶	"**begin_method**" *id* "(" *id_list* ")" "**begin_var**" *id_list* "**end_var**" *instruction_list* "**end_method**"
instruction_list	⟶	*instruction instruction_list* — *instruction*
instruction	⟶	*if* — *while* — *assign* "**;**" — *method_call* "**;**" — *end_if* "**;**" — *launch* "**;**"
if	⟶	"**if**" *expr* "**then**" *instruction_list* "**else**" *instruction_list* "**end_if**"
while	⟶	"**while**" *expr* "**do**" *instruction_list* "**end_while**"
assign	⟶	*id* "**:=**" *expr*
method_call	⟶	*id* "(" *id_list* ")"
when	⟶	"**when**" *expr* "**then**" *instruction_list* "**end_when**"
launch	⟶	"**launch**" *method_call*
expr	⟶	*expr bin_op expr* — *un_op expr* — "(" *expr* ")" — *id* — *cst*
bin_op	⟶	"**+**" — "**-**" — "*****" — "**/**" — "**¡**" — "**¿**" — "**¡¿**" — "**¡=**" — "**¿=**" — "**and**" — "**or**"
un_op	⟶	"**-**" — "**not**"

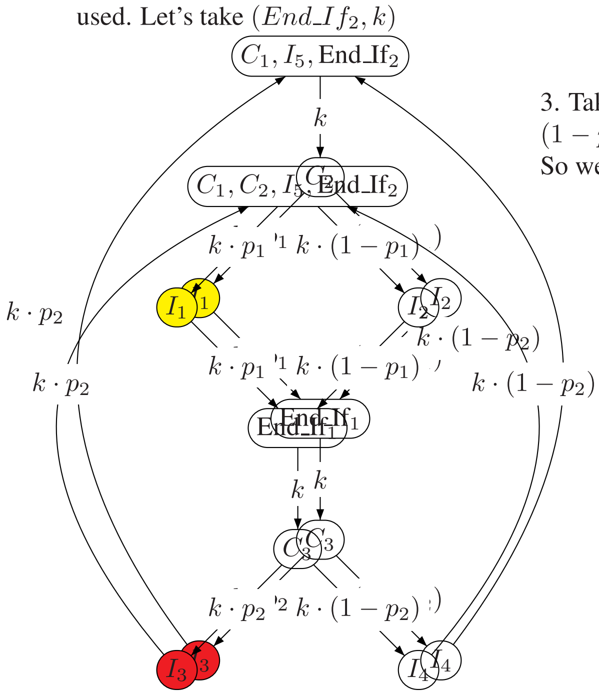
B Non reducible graphs



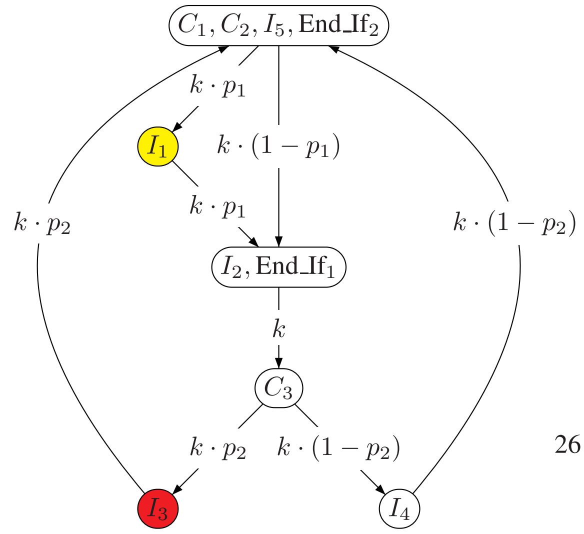


1. Take node C_1, edges : $(I_5, k), (C_2, k)$. The condition is fulfilled : either one of the edges can be used to merge C_1 with respectively I_5 or C_2. Let's take (I_5, k).

2. Take node C_1, I_5, edges : $(End\_If_2, k), (C_2, k)$. Condition fulfilled : either one of the edges can be used. Let's take (End_If_2, k)

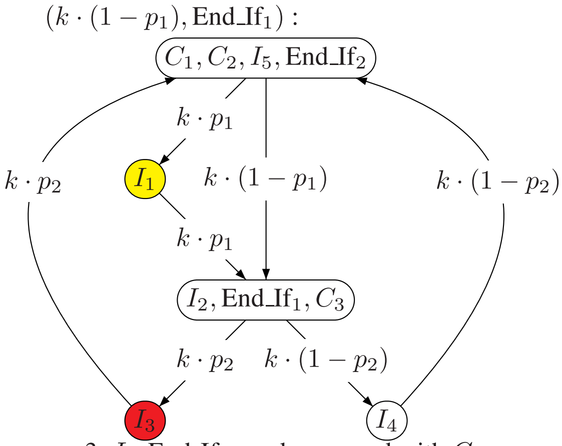


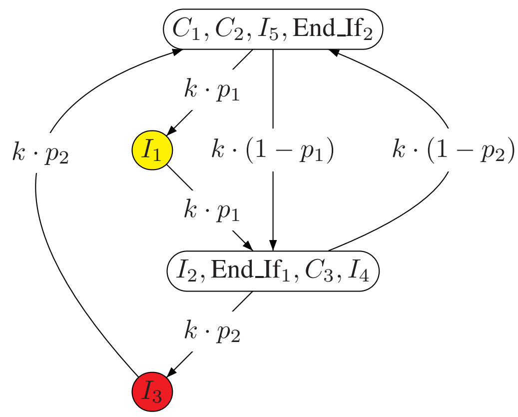
3. Take node $C_1, I_5, \text{End\_If}_2$, edges : $(C_2, k), (I_3, k \cdot p_2), (I_4, k \cdot (1-p_2))$. The condition is fulfilled : $k \geq k \cdot p_2 + k \cdot (1 - p2)$. So we can reduce using the edge (C_2, k)



1. Take node $C_1, C_2, I_5, \text{End\_If}_2$, edges $(k \cdot p_2, I_3), (k \cdot p_1, I_1), (k \cdot (1 - p_1), I_2), (k \cdot (1 - p_2), I_4)$. No edge can be found such that its weight is greater than or equal to the sum of the other edges.

2. Take node I_2, edges $(k \cdot (1 - p_1), C_1, C_2, I_5, \text{End\_If}_2), (k \cdot (1 - p_1), \text{End\_If}_1)$ let's reduce using $(k \cdot (1 - p_1), \text{End\_If}_1)$:

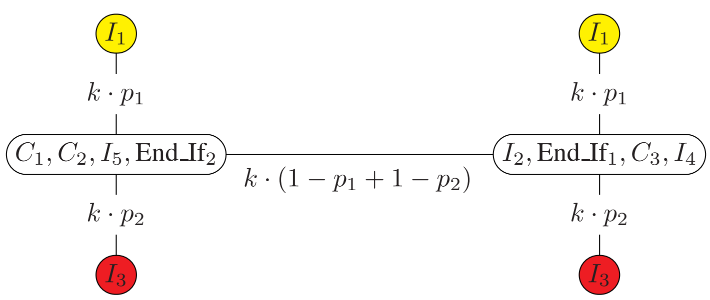




3. $I_2, \text{End\_If}_1$ can be merged with C_3

4. I_4 can be merged with $I_2, \text{End\_If}_1, C_3$

Or represented otherwise :

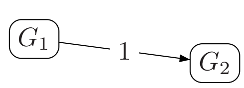


Note that for this graph to be not reducible, we must have that

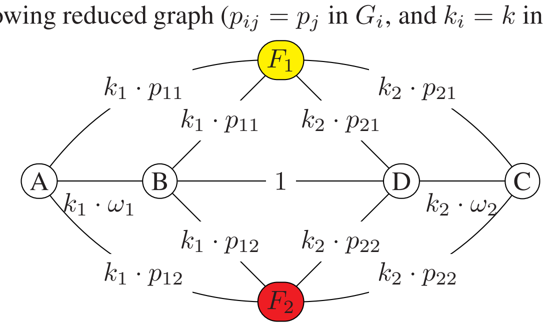
$$\begin{cases} p_1 < p_2 + (1 - p_1 + 1 - p_2) \\ p_2 < p_1 + (1 - p_1 + 1 - p_2) \\ 1 - p_1 + 1 - p_2 < p_1 + p_2 \end{cases} \equiv \begin{cases} p_1 < 1 & \text{(always satisfied)} \\ p_2 < 1 & \text{(always satisfied)} \\ p_1 + p_2 > 1 \end{cases}$$

If this third condition does not hold ($p_1 + p_2 \leq 1$), then we can merge the two non colored nodes into one node that then take the color of I_1 if $p_1 > p_2$, the color of I_3 otherwise. Note that this is the best solution, even in the case ($p_1 + p_2 > 1$)

Using the same graph chained in sequence a 2-sequence G_1, G_2 (C_1 of G_1 connected to C_1 of G_2 with weight 1) :



This results in the following reduced graph ($p_{ij} = p_j$ in G_i, and $k_i = k$ in G_i) :



with $\omega_i = k_i \cdot (1 - p_{i1} + 1 - p_{i2})$, $F_1 = \{I_{11}, I_{21}\}$, $F_2 = \{I_{13}, I_{23}\}$, $A = \{C_{11}, C_{12}, Skip_{12}, I_{15}\}$, $B = \{I_{12}, \text{End\_If}_{11}, C_{13}, I_{14}\}$, $D = \{C_{21}, C_{22}, Skip_{22}, I_{25}\}$, $C = \{I_{22}, \text{End\_If}_{21}, C_{23}, I_{24}\}$

To obtain a non reducible graph, we have eight ($i = 1, 2$) extra constraints due to the link between node B and D of weight 1 :

$$\begin{cases} k_i.p_{i1} < k_i.\omega_i + 1 + k_i.p_{i2} \\ k_i.\omega_i < k_i.p_{i1} + 1 + k_i.p_{i2} \\ 1 < k_i.p_{11} + k_i.\omega_i + k_i.p_{i2} \\ k_i.p_{i2} < k_i.p_{11} + k_i.\omega_i + 1 \end{cases} \equiv \begin{cases} p_{i1} < 1 + \frac{1}{2.k_i} & \text{(always satisfied)} \\ p_{i1} + p_{i2} > \frac{k_i - 1}{k_i} & \text{(always satisfied, see prev constraint)} \\ k_i > \frac{1}{2} & \text{(always satisfied)} \\ p_{i2} < 1 + \frac{1}{2.k_i} & \text{(always satisfied)} \end{cases}$$

More interesting graphs can be obtained by sequencing several graphs G_i to obtain a graph H_n :

$$\boxed{G_1} \xrightarrow{1} \boxed{G_2} \xrightarrow{1} \boxed{\cdots} \xrightarrow{1} \boxed{G_n}$$

Then, using H_n, we can create even more complex colorable structures by replacing nodes of a simple control flow graph by instances of H_n.