Modal Precedence Graphs and their usage in ILOG Scheduler

Philippe Laborie

ILOG Optimization Internal Report OIR-1999-1
July 1999

Contents

List of Figures

Chapter 1

Introduction

Scheduling is positioning activities in time while respecting a set of resource capacity constraints. When doing so, one often needs to reason about the relative position of activities in order to answer questions such as:

- (A) Is activity a_i before activity a_j?
- (B) What happens if I rank activity a_i before activity a_j on a given resource ?
- (C) What happens if I rank activity a_i before activity a_j and impose that no other activity a_k can be ranked between a_i and a_j on a given resource ?
- (D) What happens if I rank a_i before a_j under the condition that they require the same resource?

Such reasoning implicitly relies on a graph structure whose vertices are activities of the schedule and whose edges represent an ordering between pairs of activities. When looking more attentively at the questions above, we see that a classical precedence graph is not sufficient to represent what we need:

- Question (B) suggest that we need a dynamic graph structure on which we can add new edges and propagate the effects of these changes;
- Question (C) expresses the need for at least two kind of temporal relations between pairs of activities:
  - a successor relation that states that activity a_j cannot be started before activity a_i has been completed[1]; and
  - a next relation that states that activity a_j cannot be started before activity a_i has been completed and no other activity requiring the same resource can start or finish between the end time of a_i end the start time of a_j[2].
- When an activity only possibly requires a resource[3], question (D) shows that we need to take into account a notion of modality: we want to be able to reason on the relative position of activities only under the assumption that they will finally require the same resource.

[1] This is the semantics of the startsAfterEnd precedence constraint in Scheduler.
[2] This is the semantics of the next variable of the Scheduler sequence constraint.
[3] As opposed to *surely requires a resource.*

This technical report describes such an extended precedence graph structure, that we call *Modal Precedence Graph* and how it has been integrated into ILOG Scheduler. The expected benefits of using *Modal Precedence Graphs* in ILOG Scheduler are:

- To improve the performance of existing constraints as for example the disjunctive or the sequence constraint. Indeed, Modal Precedence Graphs allow to incrementalize the propagation of these constraints.
- To allow new propagation algorithms based on the graph structure
- To allow the implementation of new search strategies relying on the graph structure (goals that orders pairs of activities, problem decomposition strategies, etc...)

The notations, definitions and theory of *Modal Precedence Graphs* is described in chapter 2. Chapter 3 deals with the integration of *Modal Precedence Graphs* in ILOG Scheduler and its benefits. Notice that, as often as possible, we illustrate the new concepts with very basic examples. Unless stated otherwise, everything that is described in this technical report was implemented in ILOG Scheduler 4.4.

Chapter 2

Modal Precedence Graphs

2.1 Introduction

Precedence Graph. A *Precedence Graph* is a directed graph where vertices represent activities and where edges, say from vertex v_i to vertex v_j, require that activity v_i be completed before activity v_j can start.

Precedence Graphs are a very concise and convenient way to represent a set of total orders of the vertices. The total orders on vertices that satisfy the precedences expressed in the graph are called *topological sorts* of the graph.

Example 1 *The Precedence Graph on figure 2.1 represents 5 topological sorts. These 5 topological sorts are:* $\{(v_0, v_3, v_4, v_1, v_2, v_5), (v_0, v_3, v_1, v_4, v_2, v_5), (v_0, v_3, v_1, v_2, v_4, v_5), (v_0, v_1, v_3, v_2, v_4, v_5), (v_0, v_1, v_3, v_4, v_2, v_5)\}$

Now suppose that the vertices of the graph represent a set of activities that require the same unary resource in a schedule and that each edge on the graph represents a precedence constraint. The topological sorts of the graph represent the different possibilities to schedule the activities on the resource.

As said in the introduction, we would like to extend this classical representation in order to be able to represent two additional features that are very useful in realistic scheduling problems: *next relations* and *possibly contributing activities*.

Next Relations. In a Precedence Graph, a directed edge between two vertices (v_i, v_j) states that, in all the topological sorts of the graph, vertex v_i must be ranked before vertex v_j. In the following, we will call this relation a *successor relation*.

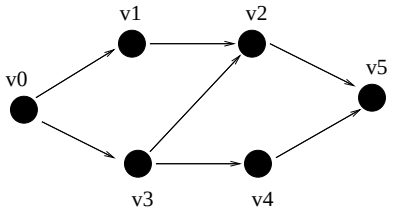

Figure 2.1: An example of Precedence Graph

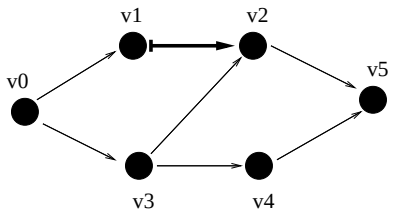

Figure 2.2: An example of Next Relation

A *next relation* between two vertices (v_i, v_j) is a relation that is stronger than a *successor relation*. It states not only that vertex v_i must be ranked before vertex v_j in all the topological sorts of the graph but also that no other vertex v_k can be ranked between v_i and v_j in any topological sort; that is, vertex v_j is next to vertex v_i in all the topological sorts of the graph.

Example 2 *In the graph of figure 2.2, the edges $\rightarrow$ represent a successor relation while the edge $\hookrightarrow$ (between v_1 and v_2) represents a next relation. Although very similar to the graph in example 1, this graph has only 2 topological sorts. These 2 topological sorts are:* $\{(v_0, v_3, v_4, v_1, v_2, v_5), (v_0, v_3, v_1, v_2, v_4, v_5)\}$

Possibly Contributing Vertex. Now, the second additional feature we would like to add to Precedence Graphs is related to the fact that we don't know for sure whether a given activity requires or not the resource. That is, a given activity may only *possibly* require the resource (as opposed to *surely*)[1].

The natural way to represent this is to associate a label to each vertex of the graph whose value can be either *contributes*, *not contributes* or *possibly contributes*. A vertex that is tagged *contributes* appears in all the topological sorts of the graph[2]. A vertex that is tagged *possibly contributes* may or may not appear in a topological sort of the graph. A vertex that is tagged *not contributes* does not appear in any topological sort.

Example 3 *Consider the graph on figure 2.3, all the vertices are contributors except vertex v_4 that only possibly contributes. This graph represents 7 topological sorts (5 topological sorts that contain vertex v_4 and two additional topological sorts without v_4). These 7 topological sorts are:* $\{(v_0, v_3, v_4, v_1, v_2, v_5), (v_0, v_3, v_1, v_4, v_2, v_5), (v_0, v_3, v_1, v_2, v_4, v_5), (v_0, v_1, v_3, v_2, v_4, v_5), (v_0, v_1, v_3, v_4, v_2, v_5), (v_0, v_3, v_1, v_2, v_5), (v_0, v_1, v_3, v_2, v_5)\}$

The notion of *Modal Precedence Graph* is an extension of *Precedence Graphs* that allows the expression of both *next relations* and *possibly contributing vertex*. *Modal Precedence Graphs* are more formally described in the rest of this chapter.

2.2 Theory

2.2.1 Basic Definitions and Notations

Let $C = \{ \{0\}, \{1\}, \{0, 1\} \}$

[1] In ILOG Scheduler, an activity may only possibly require a resource. This is the case when either the required capacity is a variable with 0 as minimal value, or the duration of the activity is a variable with 0 as minimal value or the resource constraint is meta-posted.

[2] In a classical Precedence Graph, all the vertices are implicitly considered as sure contributors.

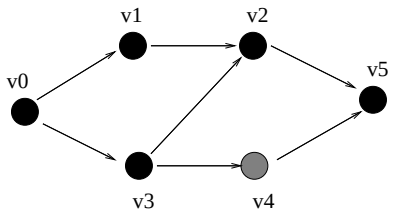

Figure 2.3: An example of Possibly Contributing Vertex

Definition 1 (Modal Precedence Graph) *A Modal Precedence Graph (MPG) is a t-uple* $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ *where:*

- V *is the set of vertices;*
- $\mathcal{C}$ *is the* **contribution function** *of the graph, that is, a function* $V \rightarrow C$*. For a vertex* $v \in V$*:*
  - $\mathcal{C}(v) = \{0\}$ *means that* v *does not contribute*
  - $\mathcal{C}(v) = \{1\}$ *means that* v *contributes*
  - $\mathcal{C}(v) = \{0, 1\}$ *means that* v *may or may not contribute to the graph*
- $\mathcal{S} \subset V \times V$ *is the set of* **successor edges** *of the graph*
- $\mathcal{N} \subset S$ *is the set of* **next edges** *of the graph*

The special case of a classical Precedence Graph is obtained when $\mathcal{N} = \emptyset$ and C is the constant function $\forall v \in V, \mathcal{C}(v) = \{1\}$.

Example 4 *According to the graphical conventions already introduced in the introduction of this chapter, the following MPG* $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ *is represented on figure 2.4.*

- $V = \{v_0, v_1, v_2, v_3, v_4, v_5\}$
- $\mathcal{C} : V \rightarrow C$ *such that:*
  - $\mathcal{C}(v_0) = \{1\}$
  - $\mathcal{C}(v_1) = \{1\}$
  - $\mathcal{C}(v_2) = \{1\}$
  - $\mathcal{C}(v_3) = \{1\}$
  - $\mathcal{C}(v_4) = \{0, 1\}$
  - $\mathcal{C}(v_5) = \{1\}$
- $\mathcal{S} = \{(v_0, v_1), (v_0, v_3), (v_1, v_2), (v_3, v_2), (v_3, v_4), (v_2, v_5), (v_4, v_5)\}$
- $\mathcal{N} = \{(v_1, v_2)\}$

As for Precedence Graphs, MPG can be seen as a compact representation of a set of total orders on the graph vertices. It is the notion of topological sort of a MPG .

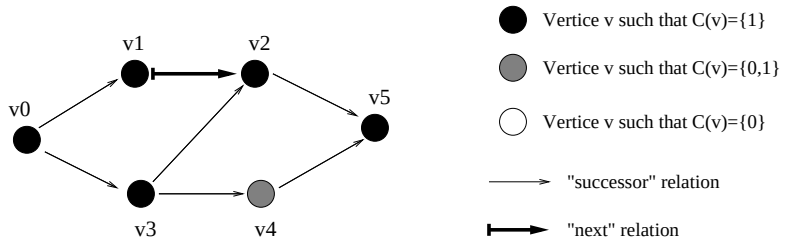

Figure 2.4: An example of Modal Precedence Graph

Definition 2 (Topological sort) *Let:*

- $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ *be a MPG where* $V = \{v_0, v_1, ..., v_n\}$.
- $\pi = (v_{\sigma(0)}, v_{\sigma(1)}, ..., v_{\sigma(k)})$ *a sublist of* V *that is:*
  1. $k \leq n$,
  2. $\forall i \in [0, k], \sigma(i) \in [0, n]$ *and*
  3. $\forall i, j \in [0, k](i \neq j) \Rightarrow (\sigma(i) \neq \sigma(j))$.

  $i \in [0, k]$ *is said to be the position of vertex* $v_{\sigma(i)}$ *in the list* π *and is also denoted* $pos(v_{\sigma(i)}, \pi)$.

π *is said to be a* **topological sort** *of* G *if and only if:*

1. $\forall v \in V, (v \in \pi) \Rightarrow (1 \in \mathcal{C}(v))$
2. $\forall v \in V, (v \notin \pi) \Rightarrow (0 \in \mathcal{C}(v))$
3. $\forall (v, w) \in \mathcal{S}, (v \in \pi \; and \; w \in \pi) \Rightarrow pos(w, \pi) > pos(v, \pi)$
4. $\forall (v, w) \in \mathcal{N}, (v \in \pi \; and \; w \in \pi) \Rightarrow pos(w, \pi) = pos(v, \pi) + 1$

Item (1) of the definition states that a vertex that appear in a topological sort cannot be tagged as not contributor on the graph. Item (2) states that if a vertex does not appear in a topological sort, then this vertex is not tagged contributor on the graph. Item (3) imposes that, if there exists a successor relation between two vertices on the graph, this order must be respected in any topological sort where they both appear. Finally, item (4) states that, if there exists a next relation between two vertices on the graph, these two vertices are constraint to be one next to the other in any topological sort where they both appear.

Example 5 *Consider the MPG of figure 2.4. This MPG represents 3 topological sorts:* $\{(v_0, v_3, v_4, v_1, v_2, v_5), (v_0, v_3, v_1, v_2, v_4, v_5), (v_0, v_3, v_1, v_2, v_5)\}$

If G is a MPG , we will denote $\Pi(G)$ the set of all the topological sorts of G [3].

Given the notion of topological sort, we can now easily define the notions of *consistence* of a MPG and of *dominance* and *equivalence* between MPG .

[3] It is clear that $\Pi(G)$ is a finite set. The biggest set $\Pi(G)$ is obtained for a MPG $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ such that $\mathcal{S} = \mathcal{N} = \emptyset$ and $\mathcal{C}$ is the constant function with value $\{0, 1\}$. In that case, if $n = |V|$, $|\Pi(G)| = \Sigma_{i=0}^{n} \frac{n!}{(n-i)!}$.

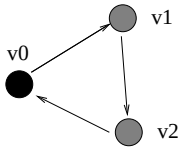

Figure 2.5: An example of Cyclic Modal Precedence Graph

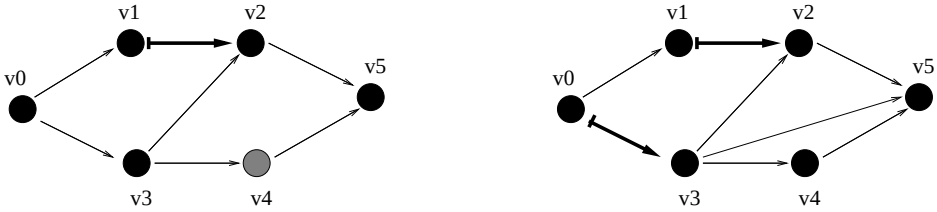

Figure 2.6: An example of Dominance

Definition 3 (Consistence) *A MPG G is said to be consistent if and only if* $\Pi(G) \neq \emptyset$.

It is important to notice that, unlike for classical Precedence Graphs, a cyclic MPG can be consistent. This is illustrated by the following example.

Example 6 *The MPG G on figure 2.5 is consistent. Indeed, we have* $\Pi(G) = \{(v_0), (v_0, v_1), (v_2, v_0)\}$.

Definition 4 (Dominance) *Let* $G_1 = (V, \mathcal{C}_1, \mathcal{S}_1, \mathcal{N}_1)$ *and* $G_2 = (V, \mathcal{C}_2, \mathcal{S}_2, \mathcal{N}_2)$ *be two MPG . G_2 is said to* **dominate** G_1 *if and only if:*

- $\mathcal{S}_1 \subset \mathcal{S}_2$
- $\mathcal{N}_1 \subset \mathcal{N}_2$
- $\forall v \in V, \mathcal{C}_2(v) \subset \mathcal{C}_1(v)$

Notice that, intuitively, a MPG G_2 dominates G_1 if G_2 imposes more constraints than G_1 that is, additional successor relations, additional next relations or more precise contribution label for a given vertex. The following theorem is a direct consequence of the definition of topological sorts:

Theorem 1 *Let G_1 and G_2 be two MPG . If G_2 dominates G_1 then,* $\Pi(G_2) \subset \Pi(G_1)$.

Example 7 *On figure 2.6, the graph on the right dominates the one on the left. The additional constraints in the graph on the right are: a next relation* (v_0, v_3)*, a successor relation* (v_3, v_5)*, and vertex* v_4 *that contributes (instead of possibly contributes).*

Definition 5 (Equivalence) *Two MPG G_1 and G_2 are said to be equivalent if and only if* $\Pi(G_1) = \Pi(G_2)$.

Example 8 *The two graphs represented on figure 2.7 are equivalent. Indeed, both of them represent the same set of topological sorts* $\{(v_0, v_1, v_2, v_3), (v_0, v_1, v_3, v_2)\}$.

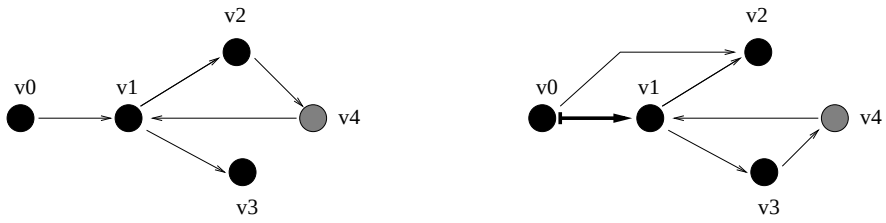

Figure 2.7: An example of Equivalence

2.2.2 Relative Positions of Vertices on the Graph

Given a MPG $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$, we will describe in this section the possible relative positions of two vertices $(v, w) \in V \times V$.

Definition 6 (Vertex Contribution) *Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . Let $v \in V$ be a vertex of G. The contribution of v to the graph G is the subset of $\{0, 1\}$ defined as follows:*

- $Contr(v) = \{0\}$ *if and only if* $\forall \pi \in \Pi(G), v \notin \pi$. *In that case, we say that v does not contribute to the graph.*
- $Contr(v) = \{1\}$ *if and only if* $\forall \pi \in \Pi(G), v \in \pi$. *In that case, we say that v contributes to the graph.*
- $Contr(v) = \{0, 1\}$ *otherwise. In that case, we say that v possibly contributes to the graph.*

Definition 7 (Relative Positions) *Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . Let $(v, w) \in V \times V$ be two vertices of G.*

- *v is said to be* **incompatible** *with w on G if and only if:*
  - $\{\pi \in \Pi(G) / (v \in \pi \; and \; w \in \pi)\} = \emptyset$

  We will denote $Incp_G(v) \subset V$ the set of vertices that are incompatible with v on G.
- *w is said to be a* **successor** *of v on G if and only if:*
  - *either v is incompatible with w*[4]
  - *or* $\forall \pi \in \Pi(G), (v \in \pi \; and \; w \in \pi) \Rightarrow (pos(w, \pi) > pos(v, \pi))$.

  We will denote $Succ_G(v) \subset V$ the set of successors of vertex v on G.
- *w is said to be a* **predecessor** *of v if and only if v is a successor of w. We will denote $Pred_G(v) \subset V$ the set of predecessors of vertex v on G.*
- *w is said to be* **next to** *v if and only if:*
  - *either v is incompatible with w*[5]

[4]The reader may be surprised by the fact that when v is incompatible with w, we still say that w is a successor of v. We use this convention for monotonicity reason: if the user has specified that w is a successor of v, this fact must remain true even if, further on, we discover that v and w are incompatible.

[5]Here again, when v is incompatible with w, we still say that w is next to v for monotonicity reason: if the user has specified that w is next to v, this fact must remain true even if, further on, we discover that v and w are incompatible.

– *or* $\forall \pi \in \Pi(G), (v \in \pi \; and \; w \in \pi) \Rightarrow (pos(w,\pi) = pos(v,\pi) + 1)$.

We will denote $Next_G(v) \subset V$ *the set of vertices that are next to vertex* v *on* G.

- w *is said to be* **previous to** v *if and only if* v *is next to* w. *We will denote* $Prev_G(v) \subset V$ *the set of vertices that are previous to vertex* v *on* G.
- w *is said to be* **possibly next to** v *if and only if:*
  - *either* v *is incompatible with* w
  - *or* $\exists \pi \in \Pi(G)/(v \in \pi) \; and \; (w \in \pi) \; and \; (pos(w,\pi) = pos(v,\pi) + 1)$

  We will denote $PNext_G(v) \subset V$ *the set of vertices that are possibly next to vertex* v *on* G.
- w *is said to be* **possibly previous to** v *if and only if* v *is possibly next to* w. *We will denote* $PPrev_G(v) \subset V$ *the set of vertices that are possibly previous to vertex* v *on* G.
- w *is said to be a* **direct successor** *of* v *on* G *if and only if:*
  - w *is a successor of* v*; and*
  - w *is possibly next to* v

  We will denote $DSucc_G(v) \subset V$ *the set of direct successors of vertex* v *on* G.
- w *is said to be a* **direct predecessor** *of* v *if and only if* v *is a direct successor of* w. *We will denote* $DPred_G(v) \subset V$ *the set of direct predecessors of vertex* v *on* G.
- w *is said to be an* **indirect successor** *of* v *on* G *if and only if:*
  - w *is a successor of* v*; and*
  - w *is not possibly next to* v

  We will denote $ISucc_G(v) \subset V$ *the set of indirect successors of vertex* v *on* G.
- w *is said to be an* **indirect predecessor** *of* v *if and only if* v *is an indirect successor of* w. *We will denote* $IPred_G(v) \subset V$ *the set of indirect predecessors of vertex* v *on* G.
- *The couple (*v*,*w*) is said to be* **unranked** *if and only if* v *is neither a successor, nor a predecessor of* w. *We will denote* $Urkd_G(v) \subset V$ *the set of vertices that are unranked with respect to* v *on* G.

Remark 1: When there is no ambiguity on the MPG , these sets will be denoted without the indexing on G. So for example $Succ(v)$ instead of $Succ_G(v)$ etc...

Remark 2: Notice the definition of these sets only relies on the set of topological sorts of a graph G. Thus, if G_0 and G_1 are two equivalent MPG , these sets are the same: $\forall v \in V, Succ_{G_0}(v) = Succ_{G_1}(v)$, etc...

Example 9 *Consider the graph of figure 2.4. We already said that this graph represents 3 topological sorts* $\Pi(G) = \{\pi_1, \pi_2, \pi_3\}$ *with* $\pi_1 = (v_0, v_3, v_4, v_1, v_2, v_5)$,$\pi_2 = (v_0, v_3, v_1, v_2, v_4, v_5)$ *and* $\pi_3 = (v_0, v_3, v_1, v_2, v_5)$. *It follows from the definitions above that:*

- v_4 *is a successor of* v_0

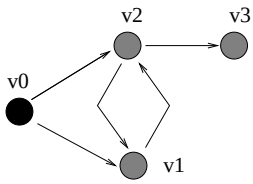

Figure 2.8: An example of Modal Precedence Graph

- v_4 *is an indirect successor of* v_0
- v_3 *is next to* v_0
- v_4 *is possibly next to* v_2 *as we have* $pos(v_4, \pi_2) = pos(v_2, \pi_2) + 1 = 4$.
- v_4 *is a direct successor of* v_3
- *(*v_2*,*v_4*) is unranked*

Example 10 *Consider the graph on figure 2.8. It represents 9 topological sorts:* $\pi_1 = (v_0)$, $\pi_2 = (v_0, v_2)$, $\pi_3 = (v_0, v_1)$, $\pi_4 = (v_0, v_2, v_3)$, $\pi_5 = (v_0, v_3)$, $\pi_6 = (v_3, v_0)$, $\pi_7 = (v_3, v_0, v_1)$, $\pi_8 = (v_0, v_3, v_1)$, $\pi_9 = (v_0, v_1, v_3)$. *According to the definitions, we have:*

- $Incp(v_1) = \{v_2\}$ *and* $Incp(v_2) = \{v_1\}$
- $Succ(v_2) = \{v_1, v_3\}$ *and* $Pred(v_2) = \{v_0, v_1\}$
- $Next(v_0) = \{v_1, v_2\}$

This example illustrate the fact that, unlike classical Precedence Graphs, on a consistent MPG , a given vertex can be at the same time successor and predecessor of another one. Furthermore, we see that several vertices can be next to a given vertex.

The following properties can be easily shown from the definitions.

Property 1 *Let* $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ *be a MPG .*

- *if* $(v, w) \in \mathcal{S}$ *then* $w \in Succ(v)$
- *if* $(v, w) \in \mathcal{N}$ *then* $w \in Next(v)$
- *if* $\mathcal{C}(v) = \{0\}$ *then* $Contr(v) = \{0\}$
- *if* $\mathcal{C}(v) = \{1\}$ *then* $Contr(v) = \{1\}$

This property shows the compatibility between the notions of *successor*, *next* and *contribution* defined in this section with the ones used when defining a MPG (successor relation, next relation, contribution function).

Property 2 (Relations between subsets of vertices) *We have the following relations between the subset of vertices.*

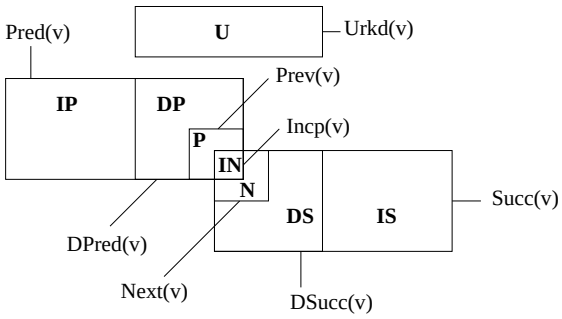

Figure 2.9: Relations Between Subsets of Vertices

1. $V = \{v\} \cup Succ(v) \cup Pred(v) \cup Urkd(v)$
2. $Succ(v) = DSucc(v) \cup ISucc(v)$
3. $Pred(v) = DPred(v) \cup IPred(v)$
4. $PNext(v) = Urkd(v) \cup DSucc(v)$
5. $PPrev(v) = Urkd(v) \cup DPred(v)$
6. $Incp(v) \subset Next(v) \subset DSucc(v)$
7. $Incp(v) \subset Prev(v) \subset DPred(v)$
8. $Incp(v) = Succ(v) \cap Pred(v)$

Most of these relations between sets are represented on figure 2.9.
Given the relations between sets, the following property holds.

Property 3 (Vertex Status) *For each $v \in V$, the following subsets of vertices form a partition of V:*

- $\{v\}$
- $U(v) = Urkd(v)$
- $IP(v) = IPred(v)$
- $DP(v) = DPred(v) \setminus Prev(v)$
- $P(v) = Prev(v) \setminus Incp(v)$
- $IN(v) = Incp(v)$
- $N(v) = Next(v) \setminus Incp(v)$
- $DS(v) = DSucc(v) \setminus Next(v)$
- $IS(v) = ISucc(v)$

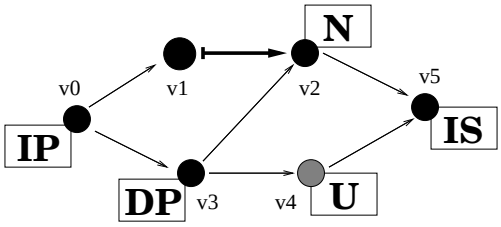

Figure 2.10: Vertex Status

Notice that all the subsets of vertices we have seen untill now can be expressed as a union of subsets of the partition defined by property 3. Thus, the relative position of a couple (v, w) on the graph is completely specified by saying which subset of the partition induced by v contains w. The symbol corresponding to this subset (U, IP, DP, P, IN, N, DS or IS) is called the status of vertex w with respect to vertex v. Furthermore, if two MPG are equivalent, they define the same relative status for their vertices. Said differently, the status of a vertex with respect to another vertex is a common feature to all the graphs belonging to the same equivalence class.

Example 11 *You can see on figure 2.10 the status of the vertices of a MPG (the one of example 4) with respect to vertex* v_1.

2.2.3 Closure

Theorem 2 (Closure) *Let* $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ *be a consistent MPG . There exists a unique MPG* $G^+ = (V, \mathcal{C}^+, \mathcal{S}^+, \mathcal{N}^+)$ *such that* G^+ *is equivalent to* G *and* G^+ *dominates all the graph in the equivalence class of* G. G^+ *is called the* **closure** *of* G.

The constructive and illustrated proof of this theorem is given in appendix A.

The following property shows some relations between the relative position of two vertices on a MPG and the edges of the closure. This property is easy to show given the construction process described in the proof of theorem 2.

Property 4 *Let* $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ *be a consistent MPG and* $G^+ = (V, \mathcal{C}^+, \mathcal{S}^+, \mathcal{N}^+)$ *its closure.*

- $Contr_G(v) = \mathcal{C}^+(v)$
- w *is incompatible with* v *on* G *if and only if* $(v, w) \in \mathcal{N}^+$ *and* $(w, v) \in \mathcal{N}^+$
- w *is a successor of* v *on* G *if and only if* $(v, w) \in \mathcal{S}^+$
- w *is next to* v *on* G *if and only if* $(v, w) \in \mathcal{N}^+$
- w *is unranked with respect to* v *on* G *if and only if* $(v, w) \notin \mathcal{S}^+$ *and* $(w, v) \notin \mathcal{S}^+$

Example 12 *As an illustration, we give on figure 2.11 the status of the vertices on the closure of the MPG of 2.10 with respect to vertex* v_1.

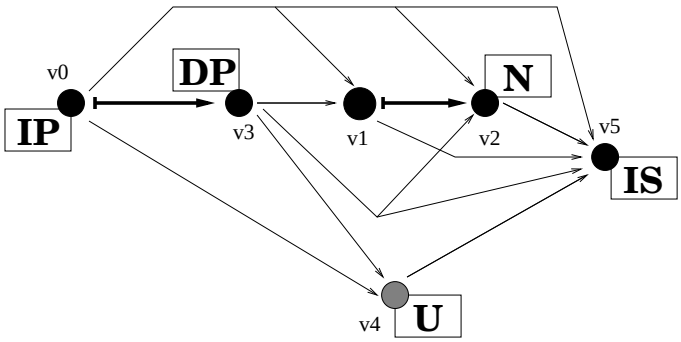

Figure 2.11: Vertex Status

2.2.4 Dynamic Modal Precedence Graphs

Graph Modifications Suppose a MPG G, we will describe in this subsection how the relative position of the vertices on this graph is changed when one modifies G by adding new constraints.

Definition 8 (MPG Modification) *Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a consistent MPG . Let $\overline{\mathcal{G}}$ be the set of all the MPG that dominate G. A* **MPG modification** *is a function α that associate to G a graph in $\overline{\mathcal{G}}$. Let v and w be two vertices of G, we distinguish four families of modifications:*

1. *$\alpha_{v\to w}$ is the modification of G such that if $G' = (V, \mathcal{C}', \mathcal{S}', \mathcal{N}') = \alpha_{v\to w}(G)$ we have:*
   - $\mathcal{C}' = \mathcal{C}$
   - $\mathcal{S}' = \mathcal{S} \cup \{(v, w)\}$
   - $\mathcal{N}' = \mathcal{N}$
2. *$\alpha_{v\hookrightarrow w}$ is the modification of G such that if $G' = (V, \mathcal{C}', \mathcal{S}', \mathcal{N}') = \alpha_{v\hookrightarrow w}(G)$ we have:*
   - $\mathcal{C}' = \mathcal{C}$
   - $\mathcal{S}' = \mathcal{S} \cup \{(v, w)\}$
   - $\mathcal{N}' = \mathcal{N} \cup \{(v, w)\}$
3. *if $\{1\} \in \mathcal{C}(v)$, α_{+v} is the modification of G such that if $G' = (V, \mathcal{C}', \mathcal{S}', \mathcal{N}') = \alpha_{+v}(G)$ we have:*
   - $\mathcal{C}'(v) = \{1\}$
   - $\forall w \neq v, \mathcal{C}'(w) = \mathcal{C}(w)$
   - $\mathcal{S}' = \mathcal{S}$
   - $\mathcal{N}' = \mathcal{N}$
4. *if $\{0\} \in \mathcal{C}(v)$, α_{-v} is the modification of G such that if $G' = (V, \mathcal{C}', \mathcal{S}', \mathcal{N}') = \alpha_{-v}(G)$ we have:*
   - $\mathcal{C}'(v) = \{0\}$
   - $\forall w \neq v, \mathcal{C}'(w) = \mathcal{C}(w)$

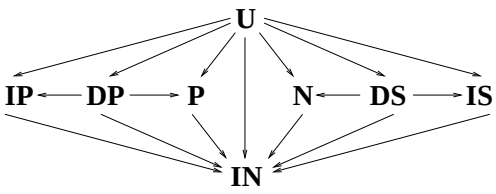

Figure 2.12: The Vertex Status Automaton

- $\mathcal{S}' = \mathcal{S}$
- $\mathcal{N}' = \mathcal{N}$

Suppose that G is a MPG and α a possible modification of G. The next paragraph will describe how the relative position of the vertices evolves between G and $\alpha(G)$.

Delta-Sets of Nodes

Property 5 *Suppose that $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ is a MPG and α a possible modification of G. Let (v, w) be a couple a vertices and $status_G(v, w) \in \{U, IP, DP, P, IN, N, DS, IS\}$ be the status that describe the relative position of (v, w) on G. Figure 2.12 shows the only possible transitions between $status_G(v, w)$ and $status_{\alpha(G)}(v, w)$.*

This property is easy to prove thanks to the fact that $\Pi(\alpha(G)) \subset \Pi(G)$.

Definition 9 (Delta-Set) *Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG , α a possible modification of G and v a vertex of G. If X is a relative position of vertices ($X \in \{U, IP, DP, P, IN, N, DS, IS\} \cup \{Pred, Succ, DPred, DSucc, PPrev, PNext, Prev, Next\}$ or any combination of these subsets), we will denote:*

- $\Delta_\alpha^- X(v) = X_G(v) \setminus X_{\alpha(G)}(v)$. *$\Delta_\alpha^- X(v)$ represents the subset of vertices whose relative position with respect to v was X before to apply the modification α and whose relative position with respect to v is no more X after applying the modification.*
- $\Delta_\alpha^+ X(v) = X_{\alpha(G)}(v) \setminus X_G(v)$. *$\Delta_\alpha^+ X(v)$ represents the subset of vertices whose relative position with respect to v is X after applying the modification α and whose relative position with respect to v was not X before applying the modification.*

Example 13 *Suppose the MPG G of figure 2.10. Now suppose we apply to this graph the modification $\alpha_{v_1 \to v_4}$. We will obtain a graph whose closure is described on figure 2.13. We also give on this figure the status of the vertices with respect to vertex v_1. In particular, notice that the status of v_4 with respect to v_1 has changed from U (cf. figure 2.11) to IS. Thus, we have $v_4 \in \Delta_\alpha^- U(v_1)$ and $v_4 \in \Delta_\alpha^+ IS(v_1)$*

The vertex status automaton represented on figure 2.12 gives the following basic relations between delta-sets:

Property 6 *Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG , α a possible modification of G and v a vertex of G.*

- $\Delta_\alpha^+ U(v) = \emptyset$

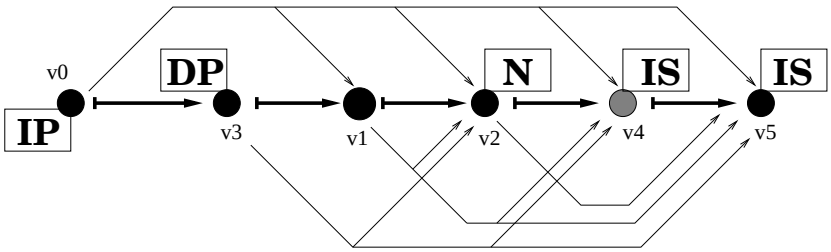

Figure 2.13: An example of Change of Status

- $\Delta_\alpha^- U(v) \subset \Delta_\alpha^+ IP(v) \cup \Delta_\alpha^+ DP(v) \cup \Delta_\alpha^+ P(v) \cup \Delta_\alpha^+ IN(v) \cup \Delta_\alpha^+ N(v) \cup \Delta_\alpha^+ DS(v) \cup \Delta_\alpha^+ IS(v)$
- $\Delta_\alpha^+ IP(v) \subset \Delta_\alpha^- U(v) \cup \Delta_\alpha^- DP(v)$
- $\Delta_\alpha^- IP(v) \subset \Delta_\alpha^+ IN(v)$
- $\Delta_\alpha^+ DP(v) \subset \Delta_\alpha^- U(v)$
- $\Delta_\alpha^- DP(v) \subset \Delta_\alpha^+ IP(v) \cup \Delta_\alpha^+ P(v) \cup \Delta_\alpha^+ IN(v)$
- $\Delta_\alpha^+ P(v) \subset \Delta_\alpha^- U(v) \cup \Delta_\alpha^- DP(v)$
- $\Delta_\alpha^- P(v) \subset \Delta_\alpha^+ IN(v)$
- $\Delta_\alpha^+ IN(v) \subset \Delta_\alpha^- IP(v) \cup \Delta_\alpha^- DP(v) \cup \Delta_\alpha^- P(v) \cup \Delta_\alpha^- IN(v) \cup \Delta_\alpha^- N(v) \cup \Delta_\alpha^- DS(v) \cup \Delta_\alpha^- IS(v)$
- $\Delta_\alpha^- IN(v) = \emptyset$
- $\Delta_\alpha^+ N(v) \subset \Delta_\alpha^- U(v) \cup \Delta_\alpha^- DS(v)$
- $\Delta_\alpha^- N(v) \subset \Delta_\alpha^+ IN(v)$
- $\Delta_\alpha^+ DS(v) \subset \Delta_\alpha^- U(v)$
- $\Delta_\alpha^- DS(v) \subset \Delta_\alpha^+ IS(v) \cup \Delta_\alpha^+ N(v) \cup \Delta_\alpha^+ IN(v)$
- $\Delta_\alpha^+ IS(v) \subset \Delta_\alpha^- U(v) \cup \Delta_\alpha^- DS(v)$
- $\Delta_\alpha^- IS(v) \subset \Delta_\alpha^+ IN(v)$

2.3 Implementation Issues

In this section, we will use the notations and results we have seen in section 2.2 to describe a structure that allows to represent MPG and an algorithm to compute their incremental closure. Our goal here is to design efficient representations and algorithms on MPG that allow, at any time, to apply some modifications to the current MPG and access in $O(1)$ time complexity the relative position of any two vertices on the current MPG .

2.3.1 Implementation Structure

Requirements. Here is the list of requirements for MPG :

1. We want to be able to create a MPG by specifying a set of vertices, a contribution function, a set of successor relations and a set of next relations.
2. We want to be able to apply any modifications as defined in section 2.2.4 on a MPG .
3. We want to be able to access in $O(1)$ to the relative position of any two vertices on a MPG .
4. Given a vertex v of a MPG , we want to be able to completely scan the subsets of vertices $U(v)$, $IP(v)$, $DP(v)$, $P(v)$, $IN(v)$, $N(v)$, $DS(v)$, $IS(v)$ with a time complexity equal to the cardinality of the subset.
5. When performing a graph modification, we want to be able to notify the user about some changes on the delta-sets of vertices ($\Delta^-_\alpha X(v) \neq \emptyset$ or $\Delta^+_\alpha X(v) \neq \emptyset$).
6. When performing a graph modification, we want to be able to scan the delta-sets of vertices with a time complexity equal to the cardinality of the delta-set.

Basic Structure. As we don't want to perform any computation when we access the relative position of two vertices (point 2), it implies that we need to store this relative position in a matrix structure that gives, for a couple (v_i, v_j) the status of v_i with respect to v_j. As we want to traverse the subset of vertices with a time complexity equal to the size of the subset (point 3), we need to store every subset of the partition $\{ U(v), IP(v), DP(v), P(v), IN(v), N(v), DS(v), IS(v) \}$ as a list.

So, basically, the structure used to represent a MPG $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ where $V = \{v_0, ..., v_n\}$ is a square matrix $M_{i=1}^n$ where an element $M(i, j)$ is a t-uple $(X_{i,j}, a_{i,j}, b_{i,j})$ (see on figure 2.14).

- $X_{i,j} \in \{U, IP, DP, P, IN, N, DS, IS\}$ is the status (or relative position) of vertex v_j with respect to vertex v_i.
- $a_{i,j} \in [1, n]$ is the element previous to v_j in the list of vertex whose status with respect to v_i is equal to $X_{i,j}$ (that is, the list $X(v_i)$).
- $b_{i,j} \in [1, n]$ is the element next to v_j in the list of vertex whose status with respect to v_i is equal to $X_{i,j}$ (that is, the list $X(v_i)$).

Notice that we use double chained lists in order to allow to remove any element from any list in $O(1)$ time. This representation is illustrated on figure 2.14. This representation heavily relies on the fact that $\{ U(v), IP(v), DP(v), P(v), IN(v), N(v), DS(v), IS(v) \}$ is a partition of V as stated by property 3.

In order to limit the memory usage of MPG , an element of the matrix is stored as one word (n bits). 4 bits are used to store the status $X_{i,j}$, the other $n - 4$ bits are used to store the index $a_{i,j}$ on $\frac{n}{2} - 2$ bits and the index $b_{i,j}$ on $\frac{n}{2} - 2$ bits[6].

For a given vertex v_i, the index of the first element of each list $U(v_i)$, $IP(v_i)$, $DP(v_i)$, $P(v_i)$, $IN(v_i)$, $N(v_i)$, $DS(v_i)$, $IS(v_i)$ is given by a set of integers $HeadU_i$, $HeadIP_i$, $HeadDP_i$, $HeadP_i$, $HeadIN_i$, $HeadN_i$, $HeadDS_i$, $HeadIS_i$.

[6]Therefore, on a 32 bits machine for example, the index $a_{i,j}$ and $b_{i,j}$ are stored on 14 bits. Notice that it limits the size of MPG to $2^{14} = 16384$.

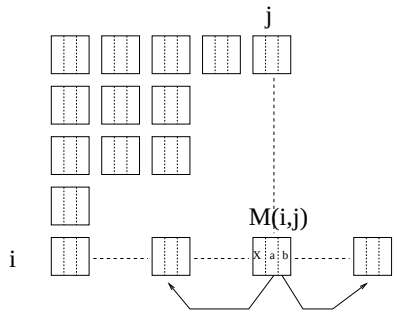

Figure 2.14: Representing a Modal Precedence Graph as a Matrix

Example 14 *The table below gives the representation of the matrix associated with the MPG of figure 2.10 (*f *stands for* first in list, l *stands for* last in list*).*

	v_0	v_1	v_2	v_3	v_4	v_5	*Heads*
v_0		*IS,f,2*	*IS,1,4*	*N,f,l*	*IS,2,5*	*IS,4,l*	*HeadIS=1, HeadN=3*
v_1	*IP,f,l*		*N,f,l*	*DP,f,l*	*U,f,l*	*IS,f,l*	*HeadIP=0, HeadN=2, etc...*
v_2	*IP,f,3*	*P,f,l*		*IP,0,l*	*U,f,l*	*DS,f,l*	*HeadIP=0, HeadP=1, etc...*
v_3	*P,f,l*	*DS,f,4*	*IS,f,5*		*DS,1,l*	*IS,2,l*	*HeadP=0, HeadDS=1, etc...*
v_4	*IP,f,l*	*U,f,2*	*U,1,l*	*DP,f,l*		*DS,f,l*	*HeadIP=0, HeadU=1, etc...*
v_5	*IP,f,1*	*IP,0,3*	*DP,f,4*	*IP,1,l*	*DP,2,l*		*HeadIP=0, HeadDP=2*

We see that with this representation, the relative position of two vertices (v_i, v_j) is directly given by $X_{i,j}$ in $O(1)$. Furthermore, one can directly iterate on any subset $X(v_i)$ as follows:
$for(j = HeadX_i; j \neq l; j = b_{i,j})$

Delta Sets. In order to represent delta-set of vertices, we split all the lists $X(v_i)$ into two sub-list: the vertices that belong to $X(v_i)$ but didn't belong to $X(v_i)$ before last modification α of the graph ($\Delta^+_\alpha X(v_i)$) and the ones that already belonged to $X(v_i)$ before last modification α. This is done simply by listing first the vertices that belongs to the delta-list ($\Delta^+_\alpha X(v_i)$) and by associating to each vertex v_i eight integers $Tail\Delta U_i$, $Tail\Delta IP_i$, $Tail\Delta DP_i$, $Tail\Delta P_i$, $Tail\Delta IN_i$, $Tail\Delta N_i$, $Tail\Delta DS_i$, $Tail\Delta IS_i$ that point to the first element of the list that does not belong to the delta-list.

Example 15 *As for example 13, suppose the MPG G of figure 2.10 on which we apply the modification* $\alpha_{v_1 \to v_4}$*. The representation of the modified MPG is given by the table under.*

	v_0	v_1	v_2	v_3	v_4	v_5	*Heads*
v_0		*IS,f,2*	*IS,1,4*	*N,f,l*	*IS,2,5*	*IS,4,l*	*HeadIS=1, HeadN=3*
v_1	*IP,f,l*		*N,f,l*	**P**,*f,l*	**IS**,*f,5*	*IS,4,l*	*HeadIP=0, HeadN=2, etc...*
v_2	*IP,f,3*	*P,f,l*		*IP,0,l*	**N**,*f,l*	*DS,f,l*	*HeadIP=0, HeadP=1, etc...*
v_3	*P,f,l*	**N**,*f,l*	*IS,f,4*		**IS**,*2,5*	*IS,4,l*	*HeadP=0, HeadN=1, etc...*
v_4	*IP,f,1*	**IP**,*0,3*	**P**,*f,l*	**IP**,*1,l*		**N**,*f,l*	*HeadIP=0, HeadP=2, etc...*
v_5	*IP,f,1*	*IP,0,3*	*DP,f,l*	*IP,1,l*	**P**,*f,l*		*HeadIP=0, HeadDP=2*

Before the modification, we had $IP(v_4) = \{v_0\}$*. After adding the successor relation* (v_1, v_4)*, we have* $IP(v_4) = \{v_0, v_1, v_3\}$*. This set is represented by the list* (v_1, v_3, v_0) *with* $Tail\Delta IP = 0$ *as* v_0 *is the first element in the list that was already here before the modification.*

The integers $Tail\Delta X_i$ allows to traverse the delta-sets of nodes $\Delta^+_\alpha X(v_i)$ with a time complexity equal to their cardinality. One can directly iterate on any subset $\Delta^+_\alpha X(v_i)$ as follows:
$for(j = HeadX_i; j \neq Tail\Delta X_i; j = b_{i,j})$

Notice that the delta-sets are accessible only once the modification has been completely performed on the graph. When these delta-sets are no more useful, they are emptied. This is easily done by setting the values of the integers $Tail\Delta X_i$ to $HeadX_i$.

Events. When applying a modification α to a MPG G, the following events are generated:

- The event $whenDirectPredecessors(v)$ is triggered as soon as the modification α causes the appearance of some new direct predecessors of v that are compatible with v. More formally when : $\Delta^+_\alpha(DP(v) \cup P(v)) \neq \emptyset$.
- The event $whenDirectSuccessors(v)$ is triggered as soon as the modification α causes the appearance of some new direct successors of v that are compatible with v. More formally when : $\Delta^+_\alpha(DS(v) \cup N(v)) \neq \emptyset$.
- The event $whenPredecessors(v)$ is triggered as soon as the modification α causes the appearance of some new predecessors of v that are compatible with v. More formally when : $\Delta^+_\alpha(IP(v) \cup DP(v) \cup P(v)) \neq \emptyset$.
- The event $whenSuccessors(v)$ is triggered as soon as the modification α causes the appearance of some new successors of v that are compatible with v. More formally when : $\Delta^+_\alpha(IS(v) \cup DS(v) \cup N(v)) \neq \emptyset$.
- The event $whenPossiblePrevious(v)$ is triggered as soon as the modification α causes the disappearance of some possible previous vertices of v that were compatible with v. More formally when : $\Delta^-_\alpha(U(v) \cup DP(v) \cup P(v)) \neq \emptyset$.
- The event $whenPossibleNext(v)$ is triggered as soon as the modification α causes the disappearance of some possible next vertices of v that were compatible with v. More formally when : $\Delta^-_\alpha(U(v) \cup DS(v) \cup N(v)) \neq \emptyset$.
- The event $whenContribution(v)$ is triggered as soon as the modification α changes a vertex from possibly contributor to contributor or not contributor. More formally when : $Contr_G(v) = \{0,1\}\ and\ (Contr_{\alpha(G)}(v) = \{0\}\ or\ Contr_{\alpha(G)}(v) = \{1\})$

It is interesting to see which events are generated on vertex v when the status of a vertex w with respect to v changes. The table under gives, for each possible transition on the vertex status automaton associated with vertex v the events that are generated on v.

Transition	*Events*
$U \rightarrow IP$	whenPredecessors whenPossiblePrevious whenPossibleNext
$U \rightarrow DP$	whenDirectPredecessors whenPredecessors whenPossibleNext
$U \rightarrow P$	whenDirectPredecessors whenPredecessors whenPossibleNext
$U \rightarrow IN$	whenPossiblePrevious whenPossibleNext
$U \rightarrow N$	whenDirectSuccessors whenSuccessors whenPossiblePrevious
$U \rightarrow DS$	whenDirectSuccessors whenSuccessors whenPossiblePrevious
$U \rightarrow IS$	whenSuccessors whenPossiblePrevious whenPossibleNext
$IP \rightarrow IN$	
$DP \rightarrow IN$	whenPossiblePrevious
$P \rightarrow IN$	whenPossiblePrevious
$N \rightarrow IN$	whenPossibleNext
$DS \rightarrow IN$	whenPossibleNext
$IS \rightarrow IN$	
$DP \rightarrow IP$	whenPossiblePrevious
$DP \rightarrow P$	
$DS \rightarrow IS$	whenPossibleNext
$DS \rightarrow N$	

Example 16 *Again, let's illustrate the notion of graph events on the MPG of figure 2.10 on which we apply the modification $\alpha_{v_1 \rightarrow v_4}$. This modification will trigger the following events:*

Vertex	*Events*	*Justification*
v_0	*No event*	
v_1	*whenSuccessor*	$\Delta^+_\alpha(IS(v_1) \cup DS(v_1) \cup N(v_1)) = \{v_4\}$
	whenPossiblePrevious	$\Delta^-_\alpha(U(v_1) \cup DP(v_1) \cup P(v_1)) = \{v_4\}$
v_2	*whenDirectSuccessor*	$\Delta^+_\alpha(DS(v_2) \cup N(v_2)) = \{v_4\}$
v_3	*whenPossibleNext*	$\Delta^-_\alpha(U(v_3) \cup DS(v_3) \cup N(v_3)) = \{v_4\}$
v_4	*whenDirectPredecessor*	$\Delta^+_\alpha(DP(v_4) \cup P(v_4)) = \{v_2\}$
	whenPredecessor	$\Delta^+_\alpha(IP(v_4) \cup DP(v_4) \cup P(v_4)) = \{v_1, v_2\}$
	whenPossiblePrevious	$\Delta^-_\alpha(U(v_4) \cup DP(v_4) \cup P(v_4)) = \{v_3\}$
	whenPossibleNext	$\Delta^-_\alpha(U(v_4) \cup DS(v_4) \cup N(v_4)) = \{v_1\}$
v_5	*No event*	

2.3.2 Algorithms for Computing Closure

Untill now, we have have introduced some definitions, described some formal properties and given the semantics of MPG . We also have described the implementation structure of MPG . It is now time to see how we can compute the closure of a MPG . In this subsection, we will describe a set of propagation rules on a MPG . A propagation rule ρ allows to transform a MPG G into a new MPG $\rho(G)$ such that $G \equiv \rho(G)$ and $\rho(G)$ dominates G. If $\{\rho_1, ..., \rho_r\}$ is a set of propagation rules, starting from a MPG G, we say that we reach a fix point G_0 when:

1. There exists a list of propagation rules $(\rho_{\sigma(1)}, ..., \rho_{\sigma(s)})$ with possible repetition of the same rules such that $G_0 = \rho_{\sigma(s)} \circ ... \circ \rho_{\sigma(1)}(G)$ and
2. $\forall i \in [1, r], \rho_i(G_0) = G_0$

As we will show, the propagation rules we describe ensure to reach a fix point that is unique and that is the closure of the initial MPG . Our propagation rules belong to the six following patterns:

- Incompatibility Rule (RC1)
- Transitive Closure through Contributor (RS1)
- Next-Edge Closure on the Left (RS2L)
- Next-Edge Closure on the Right (RS2R)
- Next-Edge Finding (RN1)
- Non-Contributor Saturation (RN2)

Incompatibility Rule (RC1). Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . If v_i and v_j are two vertices of the MPG such that: $(v_i, v_j) \in \mathcal{S}$ and $(v_j, v_i) \in \mathcal{S}$ and $\mathcal{C}(v_j) = \{1\}$ then apply to G the modification α_{-v_i}.

Transitive Closure through Contributor (RS1). Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . If v_i and v_j are two vertices of the MPG such that: $\exists v_k \in V / (v_i, v_k) \in \mathcal{S}, (v_k, v_j) \in \mathcal{S}\ and\ \mathcal{C}(v_k) = \{1\}$ then apply to G the modification $\alpha_{v_i \to v_j}$.

Next-Edge Closure on the Left (RS2L). Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . If v_i and v_j are two vertices of the MPG such that: $\exists v_k \in V / (v_k, v_i) \in \mathcal{N}, (v_k, v_j) \in \mathcal{S}\ and\ \mathcal{C}(v_k) = \{1\}$ then apply to G the modification $\alpha_{v_i \to v_j}$.

Next-Edge Closure on the Right (RS2R). Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . If v_i and v_j are two vertices of the MPG such that: $\exists v_k \in V / (v_j, v_k) \in \mathcal{N}, (v_i, v_k) \in \mathcal{S}\ and\ \mathcal{C}(v_k) = \{1\}$ then apply to G the modification $\alpha_{v_i \to v_j}$.

Next-Edge Finding (RN1). Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . If v_i and v_j are two vertices of the MPG such that: $\forall v_k \in V / (k \notin \{i, j\}), (v_k, v_i) \in \mathcal{S}\ or\ (v_j, v_k) \in \mathcal{S}$ then apply to G the modification $\alpha_{v_i \hookrightarrow v_j}$.

Non-Contributor Saturation (RN2). Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . If v_i is a vertex of the MPG such that $\mathcal{C}(v_i) = \{0\}$ then apply to G the set of modifications $\alpha_{v_i \hookrightarrow v_j}$ for $j \in [1, n]$[7].

The first five patterns are represented on figure 2.15. On this figure, the result of the propagation of each pattern of rule is shown on the right. A vertex is represented by a square if the rule does not impose conditions on its contribution label.

The following theorem states that our set of propagation rules allows to compute the closure of any MPG .

[7]This rule is not used in practise as we prefer to consider that a not contributing vertex is out of the graph. We give this rule here just to stick with the formal definition of the closure of a MPG in which, for homogeneity reasons, all not contributing vertex v_i is supposed to be linked with all the other vertices by a symmetrical next edge.

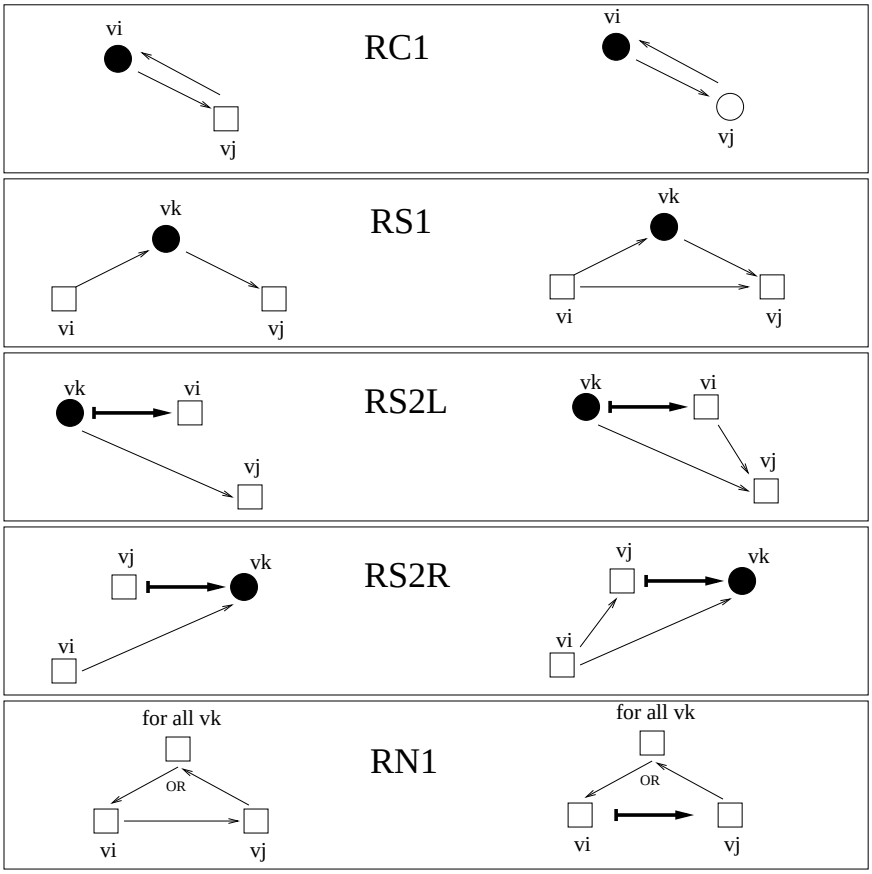

Figure 2.15: Propagation Rule Patterns

Theorem 3 (Closure) *Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a MPG . The following set of propagation rules: $\{RC1(v,w), RS1(v,w), RS2L(v,w), RS2R(v,w), RN1(v,w)\}_{(v,w)\in V\times V} \cup \{RN2(v)\}_{v\in V}$ allows to reach a unique fix point. This fix point is the closure of G.*

It is easy to see that the fix point reached with the propagation rules is a MPG equivalent to G[8]. Notice that the theorem gives an algorithm to compute the closure of a MPG by applying propagation rules till the fix point (closure) is reached. As the number of iteration is polynomial (the maximal number of new edges one can add on the MPG is n^2, the maximal number of propagations on the contribution label of nodes is n), and as the propagation of each rule is itself polynomial (in $O(n^3)$ in worst case) the theorem has the following corollary:

Corollary 1 (Closure Complexity) *Computing the closure of a MPG can be done in polynomial time.*

Example 17 *Consider the MPG on figure 2.10.*

- *Applying $RS2R(v_3, v_1)$ leads to discovering the successor relation $v_3 \rightarrow v_1$ on the graph.*
- *Applying $RS1(v_i, v_j)$ to compute the transitive closure leads to discovering the following successor relations: $v_0 \rightarrow v_2$, $v_0 \rightarrow v_4$, $v_0 \rightarrow v_5$, $v_3 \rightarrow v_5$, $v_1 \rightarrow v_5$.*
- *Applying $RN1(v_0, v_3)$ leads to discovering the next relation $v_0 \hookrightarrow v_3$.*

Finally, we obtain the transitive closure of the MPG as described on figure 2.11.

Incremental Closure. We see that applying the propagation rules from scratch and in a very naive way leads to a closure algorithm in $O(n^5)$. This is of course not realistic. The closure algorithm that has been implemented is an incremental algorithm that propagates a MPG modification. A modification α_{+v} is propagated by calling the function $propagateContribute(v)$; a modification α_{-v} is propagated by calling the function $propagateNotContribute(v)$; a modification $\alpha_{v\rightarrow w}$ is propagated by calling the function $propagateSuccessor(v, w)$ and a modification $\alpha_{v\hookrightarrow w}$ is propagated by calling the function $propagateNext(v, w)$. The algorithm corresponding to these functions is described below.

```
1.  propagateNotContribute(v) {
2.    if (C(v)=={0})
3.      return;
4.    if (C(v)=={1})
5.      exit(''inconsistent graph'');
6.    C(v)  = {0};
7.    emptySubsets(v);
8.    for (w in V)
9.      removeFromItsSubset(w,v);
10.   triggerEvents(v, whenContribution);
11. }
```

The only action of the function $propagateNotContribute(v)$ consists in emptying the subsets of vertices associated with vertex v in line 7 and deleting v from all the subsets of vertices where it is referenced in line 9.

[8]The proof that this graph is the closure of G has still not been formalized

```
1.  propagateContribute(v) {
2.    if (C(v)=={1})
3.      return;
4.    if (C(v)=={0})
5.      exit(‘‘inconsistent graph’’);
6.    for (w in IN(v))
7.      stackModif(ModifStack, (NotContribute, w));
8.    for (p in Pred(v))
9.      for (s in Succ(v))
10.       markIndirectSuccessor(p,s);
11.   for (n in N(v))
12.     for (w in Succ(v))
13.         stackModif(ModifStack, (Successor, n, w));
14.   for (p in P(v))
15.     for (w in Pred(v))
16.         stackModif(ModifStack, (Successor, w, p));
17.   triggerEvents(v, whenContribution);
18.   propagateStack(ModifStack);
19. }
```

The lines 6-7 of the function $propagateContribute(v)$ represent the incremental part of the propagation rule $RC1$ when node v_i appear to contribute. The lines 8-10 correspond to the incremental propagation of rule $RS1$. The lines 11-13 to the incremental propagation of rule $RS2L$. The lines 14-16 to the incremental propagation of rule $RS2R$. Notice that the new successor relations as well as the not contribute modification are stored on a modification stack ($ModifStack$). The reason is that it may be dangerous to perform the modification synchronously as it may interact with the iteration on the subsets. For example, the iteration on line 14 may be perturbed by the execution of a function $propagateSuccessor(n, w)$ at line 15. The function $markIndirectSuccessor(v, w)$ will be described later. It changes the position of vertex w in the partition induced by vertex v so that w becomes an indirect successor of v and similar thing with v in the partition induced by vertex w.

```
1.  propagateSuccessor(v,w) {
2.    if (w in Succ(v))
3.      return;
4.    markDirectSuccessor(v,w);
5.    if (C(w)=={1})
6.      for (s in Succ(w))
7.        markIndirectSuccessor(v,s);
8.    if (C(v)=={1}) {
9.      if (C(w)=={1})
10.       for (p in Pred(v))
11.         for (s in Succ(w))
12.           markIndirectSuccessor(p,s);
13.     for (p in Pred(v))
14.       markIndirectSuccessor(p,w);
15.   }
16.   if (C(v)=={1})
```

```
17.     for (n in N(v))
18.       stackModif(ModifStack, (Successor, n, w))
19.   if (C(w)=={1})
20.     for (p in P(w))
21.       stackModif(ModifStack, (Successor, v, p))
22.   propagateStack(ModifStack);
23. }
```

The function $markDirectSuccessor(v, w)$ at line 4 will be described later. It changes the position of vertex w in the partition induced by vertex v so that w becomes an direct successor of v and similar thing with w in the partition induced by w. Lines 5-15 represents the incremental part of propagation rule $RS1$ when a new successor edge is added on the MPG . Lines 16-18 are related to the incremental propagation of rule $RS2L$ while lines 19-21 are related to rule $RS2R$.

```
1.  propagateNext(v,w) {
2.    if (w in N(v))
3.      return;
4.    propagateSuccessor(v,w);
5.    markN(v,w);
6.    markP(w,v);
7.    if (C(v)=={1})
8.      for (s in Succ(v))
9.        stackModif(ModifStack, (Successor, w, s))
10.   if (C(w)=={1})
11.     for (p in Pred(w))
12.       stackModif(ModifStack, (Successor, p, v))
13.   triggerEvents(v, whenPossibleNext);
14.   triggerEvents(w, whenPossiblePrevious);
15.   propagateStack(ModifStack);
16. }
```

The function $markN(v, w)$ at line 5 will be described later. It changes the position of vertex w in the partition induced by vertex v so that w becomes a next vertex of v. Similar thing with $markP(v, w)$. The rest of the function corresponds to the incremental part of propagation rules $RS2L$ and $RS2R$ when a new next edge is inserted on the MPG .

```
1.  propagateStack(ModifStack) {
2.    while (! isEmpty(ModifStack)) {
3.      modif = unstackModif(ModifStack);
4.      if (modif == (Successor,v,w))
5.        propagateSuccessor(v,w);
6.      if (modif == (NotContribute,v))
7.        propagateNotContribute(v);
8.    }
9.  }
```

The function $propagateStack(ModifStack)$ iteratively applies the modifications that have previously been stored.

```
1.  markDirectSuccessor(v,w) {
2.    if (w in DSucc(v))
3.      return;
4.    markDS(v,w);
5.    markDP(w,v);
6.    if (w in IN(w)) {
7.      if (C(v)={1}) and (C(w)={1})
8.        exit(''inconsistent graph'');
9.      if (C(v)={1})
10.       stackModif(ModifStack, (NotContribute, w));
11.     if (C(w)={1})
12.       stackModif(ModifStack, (NotContribute, v));
13.   } else {
14.     triggerEvents(v, whenDirectSuccessors,
                         whenSuccessors,
                         whenPossiblePrevious);
15.     triggerEvents(w, whenDirectPredecessors,
                         whenPredecessors,
                         whenPossibleNext);
16.   }
17. }
```

The function $markDirectSuccessor(v, w)$ change the relative status of (v, w) on the graph (lines 4-5) and perform an incremental propagation of rule $RC1$ (lines 6-12). According to the change of status of the couple (v, w), the suitable events are triggered at lines 14-16

```
1.  markIndirectSuccessor(v,w) {
2.    if (w in ISucc(v))
3.      return;
4.    bool directSucc = (w in DSucc(v));
5.    bool pred       = (w in Pred(v));
6.    markIS(v,w);
7.    markIP(w,v);
8.    if (U(v) == EmptySet) and (card(DS(v)) == 1) {
9.      vertex n = DS(v);
10.     markN(v,n);
11.     markP(n,v);
12.   }
13.   if (U(w) == EmptySet) and (card(DP(w)) == 1) {
14.     vertex p = DP(w);
15.     markN(p,w);
16.     markP(w,p);
17.   }
18.   if (w in IN(w)) {
19.     if (C(v)={1}) and (C(w)={1})
20.       exit(''inconsistent graph'');
21.     if (C(v)={1})
```

```
22.       stackModif(ModifStack, (NotContribute, w));
23.     if (C(w)={1})
24.       stackModif(ModifStack, (NotContribute, v));
25.   }
26.   if (pred) {
27.     triggerEvents(v, whenPossiblePrevious);
28.     triggerEvents(w, whenPossibleNext);
29.   } else if (dsucc) {
30.     triggerEvents(v, whenPossibleNext);
31.     triggerEvents(w, whenPossiblePrevious);
32.   } else {
33.     triggerEvents(v, whenSuccessors,
                        whenPossiblePrevious,
                        whenPossibleNext);
34.     triggerEvents(w, whenPredecessors,
                        whenPossiblePrevious,
                        whenPossibleNext);
35.   }
36. }
```

The function $markIndirectSuccessor(v, w)$ change the relative status of (v, w) on the graph (lines 6-7). This function also propagates a limited version of rule $RN1$ at lines 8-17. Basically, as soon as the possible next of a vertex v changes (because some vertex previously possible next becomes indirect successor and thus, no more possible next) a check is made to see whether all the vertices are ranked with respect to v and the vertex v has only one direct successor, in that case, this unique direct successor is a next vertex of v. The function also performs an incremental propagation of rule $RC1$ (lines 18-25). According to the change of status of the couple (v, w), the suitable events are triggered at lines 26-35.

```
1.  markIS(v,w) {
2.    if (w in IS(v)) or (w in IN(v))
3.      return;
4.    removeFromItsSubset(v,w);
5.    if (w in Pred(v))
6.      addInSubset(IN(v),w);
7.    else
8.      addInSubset(IS(v),w);
9.  }
```

The function $markIS(v, w)$ removes vertex w from the subset that described its former position with respect to v and adds it: either to the subset $IS(v)$ in the case w was not a predecessor of v, either to the subset $IN(v)$ in the case w was a predecessor of v.

```
1.  markDS(v,w) {
2.    if (w in DS(v)) or (w in IN(v))
3.      return;
4.    removeFromItsSubset(v,w);
```

```
5.    if (w in Pred(v)) or (w in IS(v))
6.      addInSubset(IN(v),w);
7.    else
8.      addInSubset(DS(v),w);
9.  }
```

The function $markDS(v, w)$ removes vertex w from the subset that described its former position with respect to v and adds it: either to the subset $DS(v)$ in the case w was not a predecessor of v, either to the subset $IN(v)$ in the case w was a predecessor of v.

```
1.  markN(v,w) {
2.    if (w in IN(v)) or (w in N(v))
3.      return;
4.    removeFromItsSubset(v,w);
5.    if (w in Pred(v)) or (w in IS(v))
6.      addInSubset(IN(v),w);
7.    else
8.      addInSubset(N(v),w);
9.  }
```

The function $markN(v, w)$ removes vertex w from the subset that described its former position with respect to v and adds it: either to the subset $N(v)$ in the case w was a possible next vertex of v, either to the subset $IN(v)$ in the case w was not a possible next vertex of v.

Remarks. Now if you carefully compare the algorithm above with the theoretical definitions, you will notice two discrepancies:

1. We don't exactly compute the closure of the MPG because, for complexity reasons, a rule weaker than $RN1$ is applied. Because of it, when some vertices does not necessarily contribute, some next-relations may not be discovered on the graph. An example is given on figure 2.16 where the algorithm will not discover the next relations $v_0 \hookrightarrow v_1$ and $v_0 \hookrightarrow v_2$. Notice that this discrepancy only concerns the deduction of new next relations when some vertices does not contribute for sure. All the new successor relations and the new contribution labels are completely discovered by the algorithm.

2. We may generate more events than expected. Indeed, when applying a given modification α, we have seen that the status of a vertex w with respect to a given vertex v may change according to the vertex status automata of figure 2.12. From a theoretical point of vue, the events generated on v because of the change of the relative position (v, w) only depend on the couple of status (status before the modification, status after the modification). We see on the algorithm that the events are generated when the relative position of two vertices changes during the propagation. But, during the propagation of a modification with the algorithm we have described, the status may change several times before to reach the final status. During the propagation of a modification, the relative status of two vertices follows a path on the vertex status automaton rather than just a transition. The events that are generated are the ones on the path rather than the ones on the transition (see table 2.3.1), and that's why we may get more events. A good exercise to understand this phenomenon as well as the algorithm itself consists in following the different steps of the algorithm that propagates the new successor relation

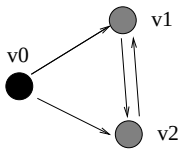

Figure 2.16: An example of Modal Precedence Graph

$v_1 \rightarrow v_4$ on the MPG of figure 2.11. We can see here that the event $whenDirectSuccessor$ will be generated on v_1 because of v_4 although after the propagation, v_4 is not a direct successor of v_1. This event is generated only because the path followed by the status of v_4 with respect to v_1 is $U \rightarrow DS \rightarrow IS$ and that v_4 has been a direct successor of v_1 transitorily during the propagation. What is important to notice is that, although some additional event may be generated, no event can be missed.

2.3.3 A Strongly Asynchronous Structure

In our implementation above ILOG Solver, three kind of functionalities can be distinguished in the API of the class that represents MPG .

- First, the functionalities related to accessing the current graph structure. These functionalities are synchronous, they includes:
  - Simple queries on the graph as "does v contributes?", "is v a successor of w?", etc...
  - Iterators on set of vertices. For example iterator on the set of direct predecessors of a given vertex v.
  - Iterators on delta-set of vertices.
- Secondly, the functionalities that define MPG modifications. It is important to notice that these modifications are performed asynchronously: the modifications are posted in the modification stack and a Solver constraint is pushed whose propagation will iteratively execute the stacked modifications by calling the suitable propagation functions on the graph. The main interest of this asynchronism is to ensure that the subsets of vertices on which one iterates will remain stable during the iteration, even if some graph modifications are posted within the iteration loop.
- Thirdly, the events on MPG vertices. These events can be associated a Solver goal that will be executed asynchronously when the event is triggered.

The figure 2.17 summarizes the global architecture of MPG implementation.

2.3.4 Memory Usage

First, it is clear that in static memory, an MPG with n vertices requires $32n + 2n^2$ words. Indeed, for any vertex v we need n reversible words (2 words) to represent the matrix row and 16 reversible words to identify the beginning of subsets of vertices (8 reversible words) and the tails of delta-lists (8 reversible words). So for example, on a 32-bits architecture, for $n = 100$, it takes $92Kbytes$. For $n = 1000$, it takes $8Mbytes$.

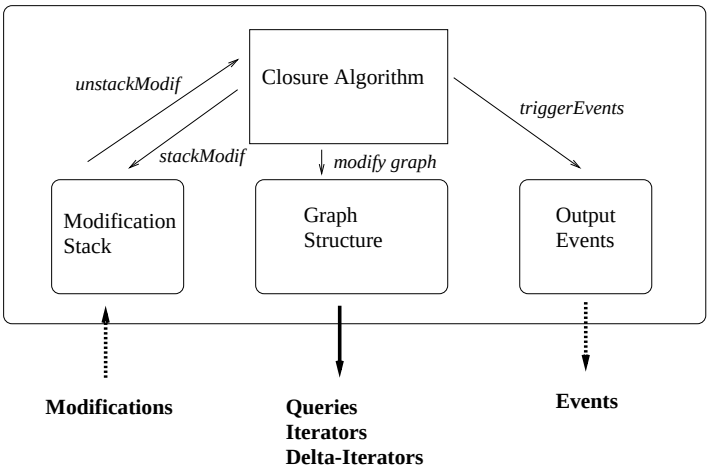

Figure 2.17: Global Architecture of Modal Precedence Graph Implementation

It is also easy to give an upper bound on the dynamic memory consumption used to store the different states of a MPG . We see on the vertex status automaton that a given vertex w of the graph can at most change 3 times of status with respect to another vertex v[9]. When the status of a vertex changes, it is removed from one sublist of vertices and added in the sublist corresponding to its new status. Removing a vertex w from a double chained list requires to save 2 reversible values (the previous and next vertex in the list). Inserting w at the head of its new list and changing its status requires to save 3 reversible values (the head of the new list, the element $M(v, w)$ itself and the next element in the new list). In the worst case, it could latter on also result in saving the value of the tail of the delta-set. Thus, the trailed memory cannot exceed $3 * 6 * 2 * n^2$. Thus, the total memory used cannot exceed $32n + 38n^2$ words. For $n = 100$, on a 32-bits architecture, it gives $1532Kbytes$. For $n = 1000$, it gives $152Mbytes$. Notice that the empirical results we made show that this upper bound on dynamic memory usage is very pessimistic.

2.3.5 Time Complexity

Worst Case Analysis. It is clear that the functions $markXXX$ are all in $O(1)$. The function $propagateNotContribute$ is in $O(n)$ in worst case (because of the iteration at line 9). The function $propagateSuccessor$ is in $O(n^3)$ in worst case. Indeed, the double loop at lines 10-11 is in $O(n^2)$ and it could be called again for all the next nodes of v (line 17) or previous nodes of w (line 20) that is n times. The function $propagateNext$ is also in $O(n^3)$ in worst case as it could execute the double loop in $propagateSuccessor$ for each next relation. The function $propagateContribute$ is also in $O(n^3)$ in worst case for the same reason. We will see in the next section that these worst-case complexities are very pessimistic ones.

Empirical Results. We have studied the experimental time performances of these functions on randomly generated MPG [10]. In our study, a random MPG can be generated by specifying four parameters $(n, \alpha, \beta, \gamma)$ where:

[9] That's the length of the longest path on the automaton.

[10] The corresponding code using ILOG Scheduler 4.4 can be found in `/nfs/schedule/techreports/OIR-1999-1/src/mpgRandom.cpp`

- n is the number of vertices of the MPG
- $\alpha \in [0,1]$ is the probability that a given vertex of the MPG contributes (the other vertices are considered to possibly contribute) that is:
  $\forall v \in V, p(Contr(v) = \{1\}) = \alpha \ \ and \ \ p(Contr(v) = \{0,1\}) = 1 - \alpha$
- $\beta \in [0,1]$ is the probability that two vertices (v,w) are ranked that is:
  $\forall (v,w) \in V \times V, p(v \in Succ(w) \ or \ v \in Pred(w)) = \beta$
- $\gamma \in [0,1]$ is the probability that two vertices (v,w) are next or previous that is:
  $\forall (v,w) \in V \times V, p(v \in N(w) \ or \ v \in P(w)) = \gamma$

The random MPG are generated by iteratively inserting new successor and next relations until the suitable density of edges is reached. Now, for each function `propagateSuccessor`, `propagateNext`, `propagateContribute`, we have studied the influence of n, α, β and γ. In the curves given in the figures below, each point corresponds to an average over 100 samples. For a given sample, we generate a random MPG G_0 with the given parameters. We also generate for this graph a list of 10 modifications (new successor, new next, or new contributor depending on the curve) that can be successively applied to G_0. The time corresponding to the sample is the time taken by the propagation of this succession of 10 MPG modifications divided by 10. Thus, it correspond to the average time of propagation of one modification and each point of the curve corresponds to an average over 1000 modifications. The times are given in μs on a $HP - UX 9000/780$ station.

The curves 2.18 shows the influence of α on the average propagation time of `propagateSuccessor` when $n = 50$ and $\beta = 0.4$. The two curves correspond to two different values of γ: 0 and 0.5. We see that in both cases the performances are the worst when $\alpha = 1$ that is when all the vertices contributes. This result can be explained by the fact that possibly contributing vertices tends to keep the propagation more local.

The curves 2.19 shows the influence of β on the average propagation time of `propagateSuccessor` when $n = 50$ and $\alpha = 1$. Here again, the two curves correspond to two different values of γ: 0 and 0.5. We see that in both cases the performances are the worst for values of β around 0.4. Looking attentively at the closure algorithm described in section 2.3.2, one would expect that the propagation time grows with β as the size of the sets $Succ(v)$, $Pred(v)$, $N(v)$ and $P(v)$ grows with β. The behavior on curves 2.19 for very dense MPG is due to some optimizations of the algorithm that are not described in this report.

The curves 2.20 shows how the average propagation time of `propagateSuccessor` scales with the size n of the graph for the worst case values of α and β (respectively 1.0 and 0.4). These curves shows that the experimental complexity of `propagateSuccessor` is something close to $O(n.\sqrt{n})$[11].

The curves 2.21 shows the influence of α on the average propagation time of `propagateNext` when $n = 50$ and $\beta = 0.4$. The two curves correspond to two different values of γ: 0 and 0.5. We see that in both cases the performances are the worst when $\alpha = 1$ that is when all the vertices contributes. Here again, this result can be explained by the fact that possibly contributing vertices tends to keep the propagation more local.

The curves 2.22 shows the influence of β on the average propagation time of `propagateNext` when $n = 50$ and $\alpha = 1$. The two curves correspond to two different values of γ: 0 and 0.5. We see that in both cases the performances are the worst for values of β around 0.4. Here again, the

[11]The curves can be approximated by the function $(0.20 * n * \sqrt{n}) + 50$.

behavior on curves 2.22 for very dense MPG is due to some optimizations of the algorithm that are not described in this report.

The curves 2.23 shows how the average propagation time of `propagateNext` scales with the size n of the graph for the worst case values of α and β (respectively 1.0 and 0.4). As for `propagateSuccessor`, the experimental complexity seems to behave in $O(n.\sqrt{n})$[12].

The curves 2.24 shows the influence of α on the average propagation time of `propagateContribute` when $n = 50$ and $\beta = 0.6$. We see here that the complexity decreases with α. To explain this behavior, we must look closely at the algorithm of function `propagateContribute`. Let's explain the behavior when $\gamma = 0$ (no next relations). For a given density of successor edges β, the average number of predecessor (and successor) of a given vertex does not depend on α: it is equal to $\beta.n$. Thus, the complexity of the double loop at lines 8-10 does not depend on α and cannot be responsible for the behavior suggested by the curves. This behavior is due to the propagation of rule $RC1$ at lines 6-7 of the algorithm. Indeed, when many vertices possibly contributes, the random generation may generate many cycles on the MPG . For $\beta = 0.6$, as the density of edges is rather high, many of these cycles are short[13]. Thus, when successively applying the 10 modifications (`setContribute`) related to a sample, it often leads to discover vertices that must not contribute and to call `setToNotContribute` at line 7. When α if greater, the density of cycles decreases and less incompatibilities need to be propagated.

The curves 2.25 shows the influence of β on the average propagation time of `propagateContribute` when $n = 50$ and $\alpha = 0.1$. The two curves correspond to two different values of γ: 0 and 0.5. We see that in both cases the propagation time grows with β until a value of β around 0.6 is reached and then it remains more or less constant. Here again, the behavior on curves 2.25 for dense MPG is due to the same optimizations evoqued before. Notice that here, anyway, these optimizations have less effect because the density of possibly contributors is high (0.9).

The curves 2.26 shows how the average propagation time of `propagateContribute` scales with the size n of the graph for the worst case values of α and β (respectively 0.1 and 0.6). The experimental complexity seems to behave in $O(n.\sqrt{n})$[14].

[12]The curves can be approximated by the function $(0.45 * n * \sqrt{n}) + 50$.

[13]For example, when $\alpha = 0.0$, it is easy to shown that the number of cycles of length k is in average $\frac{n!}{(n-k)!} \cdot \frac{\alpha^k}{2^k}$. For curves 2.24 where we have $n = 50$ and $\alpha = 0.6$ it makes 3175 cycles of length 3,

[14]The curves can be approximated by the function $(1.4 * n * \sqrt{n}) + 50$.

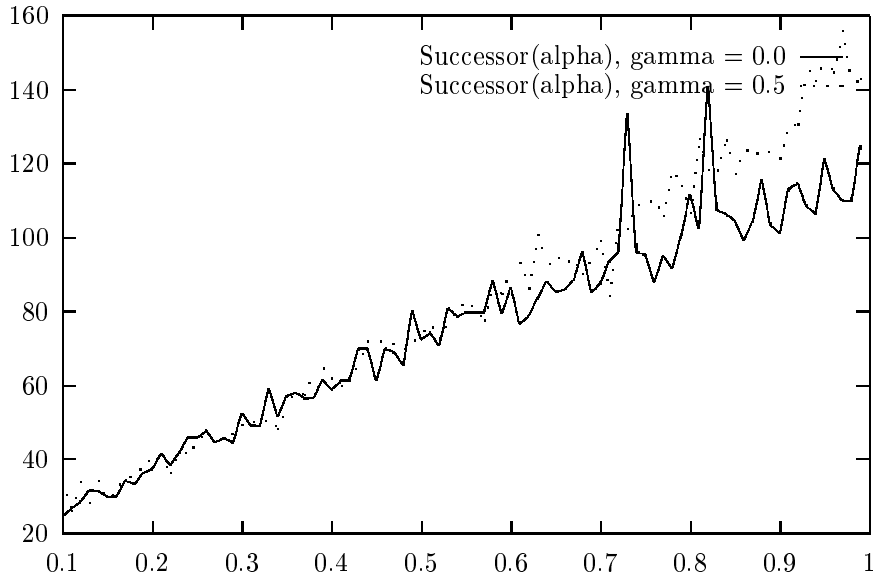

Figure 2.18: Influence of α on $propagateSetSuccessor$

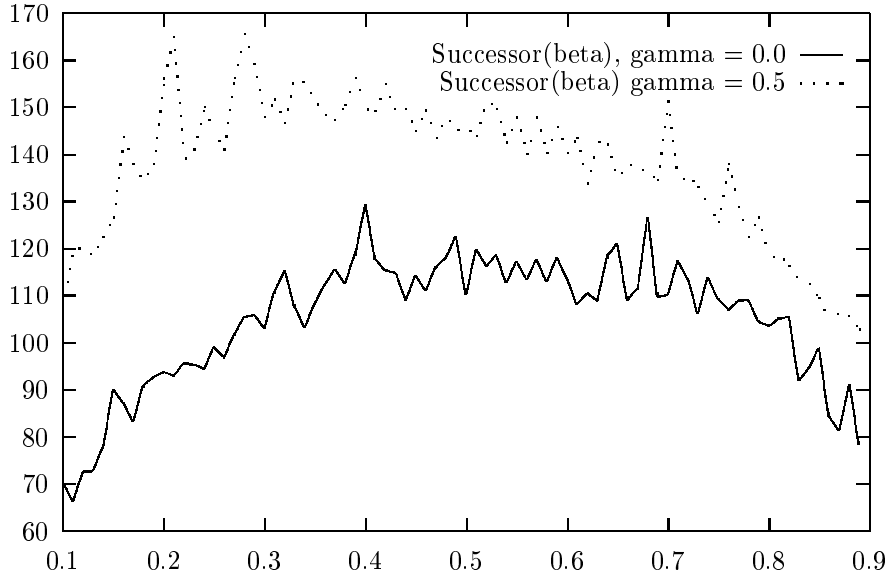

Figure 2.19: Influence of β on $propagateSetSuccessor$

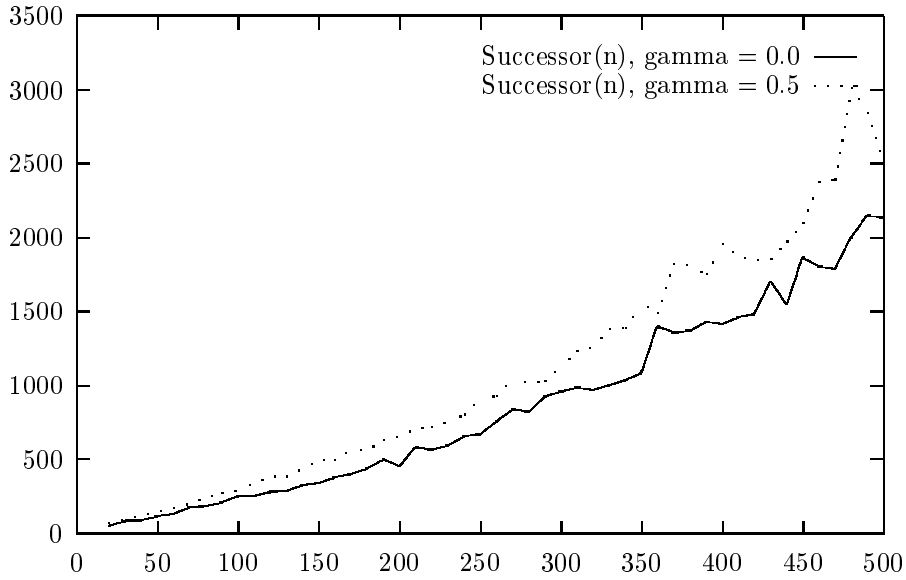

Figure 2.20: Influence of n on $propagateSetSuccessor$

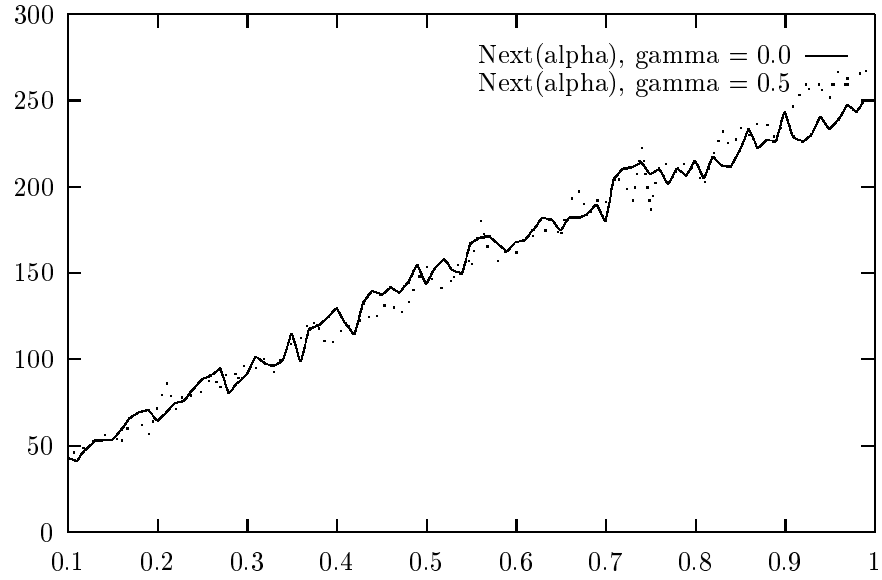

Figure 2.21: Influence of α on $propagateSetNext$

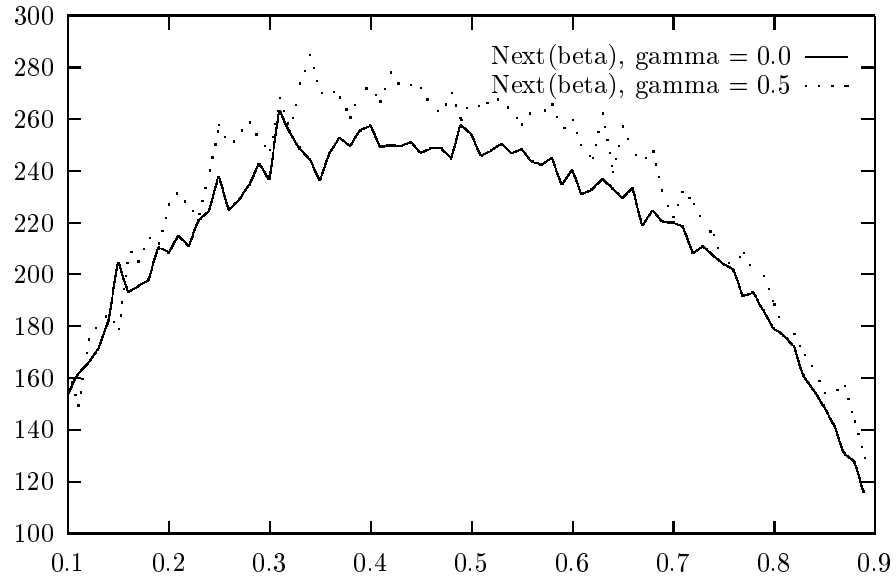

Figure 2.22: Influence of β on $propagateSetNext$

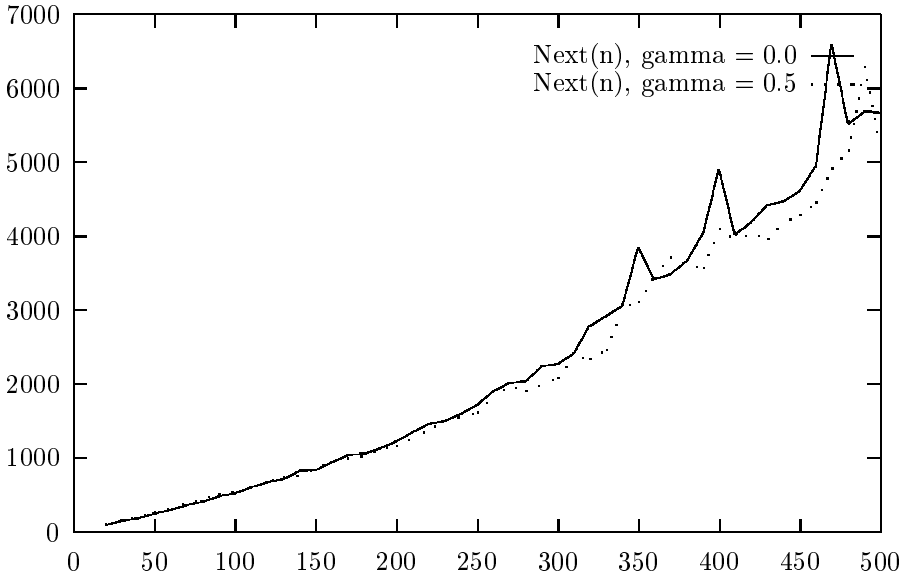

Figure 2.23: Influence of n on $propagateSetNext$

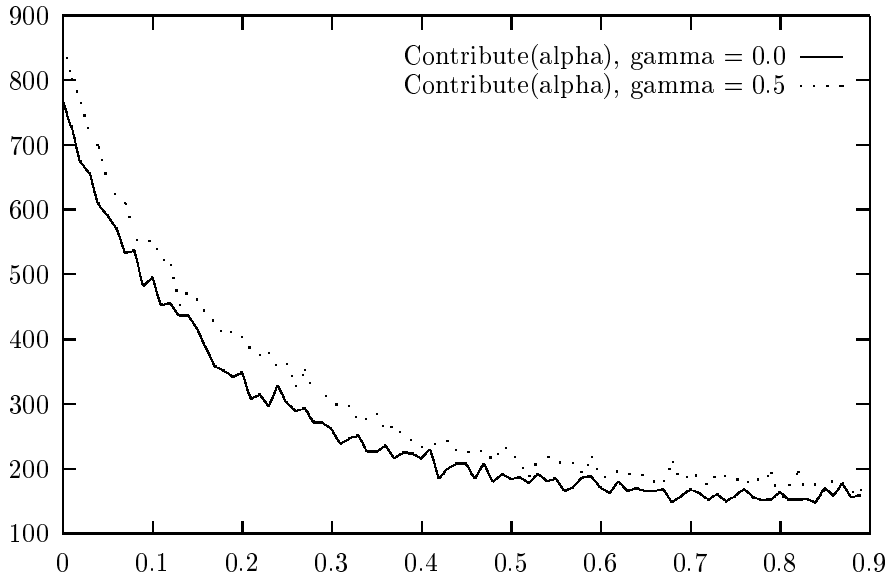

Figure 2.24: Influence of α on $propagateSetContribute$

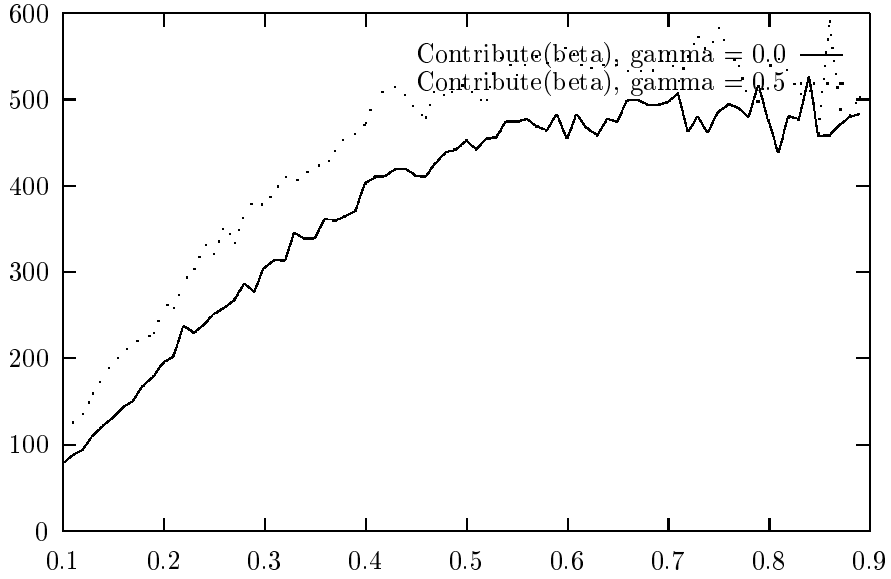

Figure 2.25: Influence of β on $propagateSetContribute$

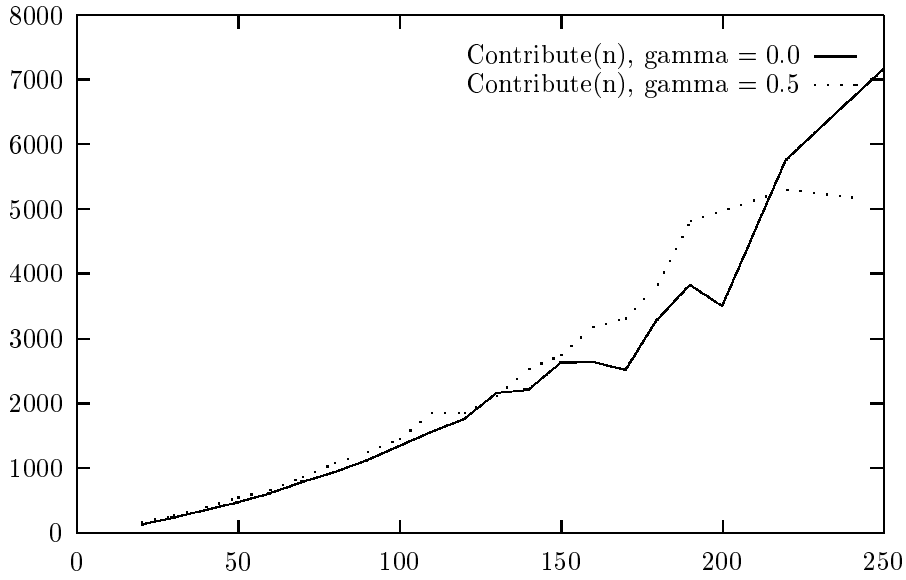

Figure 2.26: Influence of n on $propagateSetContribute$

Chapter 3

Using Modal Precedence Graphs in ILOG Scheduler

3.1 Overview

In ILOG Scheduler[1], MPG are used to represent the temporal relations between a set of activities that may require the same resource[2]. Thus, it is possible to associate with each ILOG Scheduler resource a MPG . The overview of the integration into ILOG Scheduler of the MPG on a given resource is illustrated on figure 3.1.

A constraint called *MPG Constraint* can be associated with any ILOG Scheduler resource. This constraint is in charge of creating a MPG whose vertices are the resource constraints rct on the resource. The MPG constraint is also in charge of ensuring the basic coherence between the time variables of activities (start, end, duration), the resource capacity required by the activities and the information stored on the MPG (contribution status, successor and next relations). The MPG Constraint will be described in section 3.2.

When a MPG Constraint and a Disjunctive Constraint have been defined on a resource, the disjunctive constraint scans the unranked pairs of resource constraints on the MPG in order to try to discover some new successor relations. These new successor relations are then inserted as modifications of the current MPG of the resource. The integration of MPG with the Disjunctive Constraint of ILOG Scheduler is described in section 3.3.2.

A MPG Constraint is automatically created when a Sequence Constraint is defined on a resource. In that case, the sequence constraint scans the possibly next/previous relative positions of resource constraints on the MPG to update the next/previous variables of the activities. The Sequence Constraint also discover new next relations on the graphs when some next/previous variable of an activity becomes bound. The integration of MPG with the Sequence Constraint of ILOG Scheduler is described in section 3.3.3.

We will also see in the first part of this chapter (section 3.3.1) how some Precedence Constraint between activities are automatically inserted as successor relations on the MPG of resources.

Of course, the MPG remains completely accessible to the user who can apply his own new MPG modifications (add a successor relation, a next relation or constrain the contribution of a re-

[1]We suppose in this chapter that the reader is familiar with the ILOG Scheduler predefined constraints such as: Precedence Constraint, Disjunctive Constraint, Edge-Finder Constraint or Sequence Constraint as described in [3].

[2]Notice that it is also possible, in ILOG Scheduler 4.4 to create a MPG associated with all the activities of a schedule and to have this MPG cooperate with the MPG on the resources. This kind of MPG will not be described in this report.

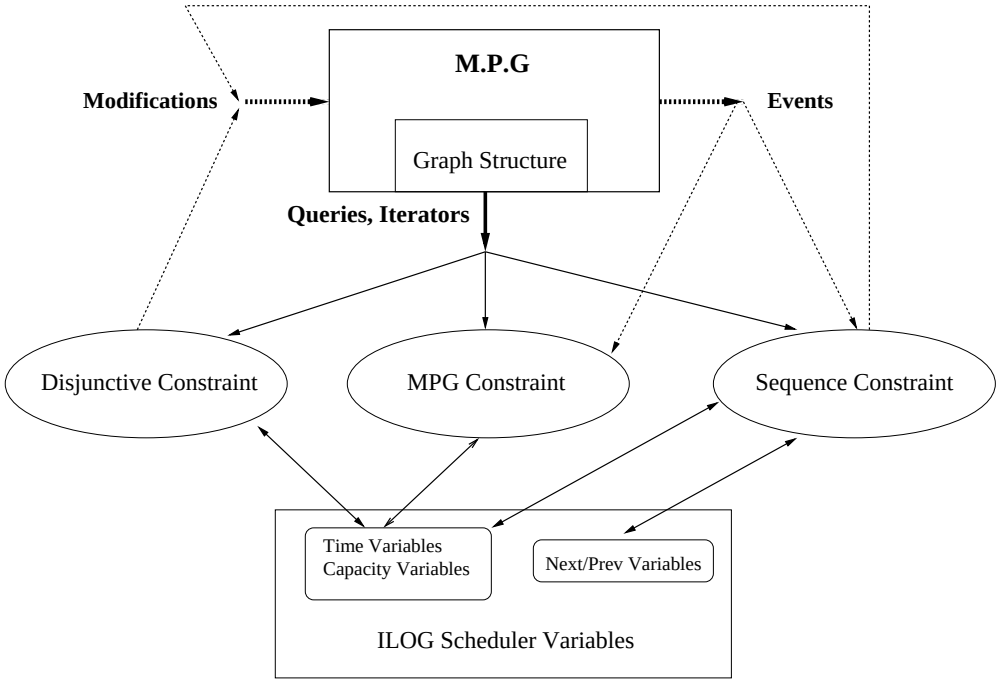

Figure 3.1: Global Overview of Architecture

source constraint), query the current MPG about the relative position of resource constraints on the resource (accessors, iterators, delta iterators) and be notified, thanks to graph events about any change in the MPG . In order to keep the API as simple as possible, we chosed to document the functionalities related to MPG on the resource and resource constraints themselves rather than to document new classes corresponding to the MPG and the MPG vertices. So for example, creating a MPG Constraint on a resource res is done by calling `res.makePrecedenceGraphConstraint()`; modifying the MPG to insert a successor relation between two resource constraints $rct1$ and $rct2$ is done by writing `rct1.setSuccessor(rct2)`, etc. The complete API can be found in [5].

3.2 MPG Constraint

3.2.1 Notations and Principle

Let R be a resource and RCT the set of resource constraints on R. We suppose that G is the MPG associated with R. Given a resource constraint rct in RCT, we denote:

- $Pred(rct)$ the set of predecessors of rct on the graph G
- $Succ(rct)$ the set of successors of rct on the graph G
- $DPred(rct)$ the set of direct predecessors of rct on the graph G
- $DSucc(rct)$ the set of direct successors of rct on the graph G
- $Urkd(rct)$ the set of resource constraints unranked with respect to rct on the graph G

- $dur(rct)$ the duration variable of the activity of rct
- $start(rct)$ the start variable of the activity of rct
- $end(rct)$ the end variable of the activity of rct
- $dmin(rct)$ the duration min of the activity of rct
- $smin(rct)$ the earliest start time of the activity of rct
- $smax(rct)$ the latest start time of the activity of rct
- $emin(rct)$ the earliest completion time of the activity of rct
- $emax(rct)$ the latest completion time of the activity of rct
- If R is a capacity resource, $cap(rct)$ will denote the capacity variable required by rct
- If R is a capacity resource, $cmin(rct)$ will denote the minimal capacity required by rct
- If R is a capacity resource, $cmax(rct)$ will denote the maximal capacity required by rct
- $TE(rct) \in \{N, A, BS, AS, BE, AE, BSAE, FSTE\}$ will denote the time extent of rct (respectively: Never, Always, Before Start, After Start, Before End, After End, Before Start And After End, From Start To End).
- $isToBeTrue(rct)$ is true if rct really needs to be satisfied that is, if rct is really posted and not just meta-posted.
- $isToBeFalse(rct)$ is true if rct really needs not to be satisfied that is, if the opposite of rct is really posted and not just meta-posted.

Whereas basic ILOG Solver constraints are constraints between a set of ILOG Solver variables, it is sometimes necessary to write more elaborated constraints that also involve some complex structures. The MPG constraint is a good illustration. Basically, the MPG constraint on a given resource R is a constraint between: (1) the start, end and duration variables of the activities that require the resource R (ILOG Solver variables), (2) The capacity variables of the resource constraints on R (ILOG Solver variables) and (3) the MPG G associated with R. As such, the MPG constraint needs to be propagated as soon as either the domain of a variable or the graph G has changed. We will see, in the following subsections how these propagations are performed.

3.2.2 Contribution Status of a Resource Constraint

As said in chapter 2, the contribution label of a vertex in a MPG represents the fact that the vertex may or may not appear in the topological sorts of the graph. In a scheduling context, it represents the fact that the activity may or may not effectively require the resource. We say that an activity effectively requires a resource if this activity affects the availability of the resource, that is, if there exists some non-empty time interval during which the availability profile of the resource is modified by the execution of the activity because the activity has required, consumed or produce a not null quantity of the resource. We see that the contribution of a resource constraint depends on the type of the resource so we need to formally define this notion for each type of resource.

Capacity Resources.

Definition 10 *A capacity resource constraint rct is said to contribute if and only if the following assertion holds:*

$$\begin{array}{ll} & (isToBeTrue(rct)) \\ \wedge & (TE(rct) \neq N) \\ \wedge & (cmin(rct) > 0) \\ \wedge & ((dmin(rct) > 0) \vee (TE(rct) \neq FSTE)) \end{array}$$

This definition says that a capacity resource constraint contributes if and only if the execution of this activity will affects for sure the availability of the resource.

Definition 11 *A capacity resource constraint rct is said to not contribute if and only if the following assertion holds:*

$$\begin{array}{ll} & (isToBeFalse(rct)) \\ \vee & (TE(rct) = N) \\ \vee & (cmax(rct) = 0) \\ \vee & ((dmax(rct) = 0) \wedge (TE(rct) = FSTE)) \end{array}$$

This last definition says in particular that a capacity resource constraint on a null duration activity with a time extent $FromStartToEnd$ cannot contribute. As well as a capacity resource constraint with a null capacity requirement.

State Resources.

Definition 12 *A state resource constraint rct is said to contribute if and only if the following assertion holds:*

$$\begin{array}{ll} & (isToBeTrue(rct)) \\ \wedge & (TE(rct) \neq N) \\ \wedge & ((dmin(rct) > 0) \vee (TE(rct) \neq FSTE)) \end{array}$$

This definition says that a state resource constraint contributes if and only if the execution of this activity will affects for sure the availability of the resource.

Definition 13 *A state resource constraint rct is said to not contribute if and only if the following assertion holds:*

$$\begin{array}{ll} & (isToBeFalse(rct)) \\ \vee & (TE(rct) = N) \\ \vee & ((dmax(rct) = 0) \wedge (TE(rct) = FSTE)) \end{array}$$

This last definition says in particular that a state resource constraint on a null duration activity with a time extent $FromStartToEnd$ cannot contribute.

Constraint Propagation. The MPG constraint maintains the coherence between the contribution of the vertices of the MPG and the resource constraints according to the definitions above. Thus, a function $checkContributionRelation$ is called each time the duration, the capacity (for capacity resource constraints only), the meta-status or the contribution of a resource constraint changes. This is done thanks to a demon attached to the events $whenDomain$ on ILOG Solver variables and events $whenContribute$ on the MPG . When executed, this demon updates, when possible, the duration, the capacity variables and the contribution of the resource constraint according to the definitions above.

3.2.3 Start and End Times of Activities

When applied to scheduling, the semantics of a successor relation between two resource constraints rct_1 and rct_2 states that whenever both rct_1 and $rct2$ contributes (so, when they both effectively affect the availability of the resource, see the above section), the activity of rct_1 is constrained to execute before the activity of rct_2 ($end(rct_1) \leq start(rct_2)$).

When a transition time is defined between rct_1 and rct_2, the MPG constraint will also propagate it. Thus, the following relation is propagated:

$$\begin{array}{ll} & Contr(rct_1) = \{1\} \\ \wedge & Contr(rct_2) = \{1\} \\ \wedge & rct_2 \in Succ(rct_1) \\ \Rightarrow & end(rct_1) + ttEndStart(rct_1, rct_2) \leq start(rct_2) \end{array}$$

$ttEndStart(rct_1, rct_2)$ is the minimal delay between the end of $rct1$ and the beginning of rct_2 taking into account the transition time between $rct1$ and $rct2$. This delay depends on the time extents of the resource constraints. The formula to compute $ttEndStart(rct_1, rct_2)$ knowing the time extents of the resource constraints and the transition time function $transTime$ is given on the table below[3]. In the most usual case when the time extents are both $FromStartToEnd$, we have $ttEndStart(rct_1, rct_2) = transTime(rct_1, rct_2)$.

$TE(rct_1)$	$TE(rct_2)$	$ttEndStart(rct_1, rct_2)$
FSTE, BE	FSTE, AS	$max(0, transTime(rct_1, rct_2))$
BS	FSTE, AS	$max(0, transTime(rct_1, rct_2) - dmin(rct_1))$
FSTE, BE	AE	$max(0, transTime(rct_1, rct_2) - dmin(rct_2))$
BS	AE	$max(0, transTime(rct_1, rct_2) - dmin(rct_1) - dmin(rct_2))$

Constraint Propagation. The relation above needs to be propagated as soon as (1) new resource constraints appear to contribute or (2) new successor relations appear on the MPG or (3) the start or end variables of activities change. Thus, some demons to propagate the relation are attached to the $whenContribution$ and $whenDirectSuccessor$ of the resource constraints in the MPG as well as on the $whenDomain$ event on start and end variables of activities.

3.2.4 Propagation Algorithm

We give in this section the propagation algorithm of the MPG constraint. The functions `checkStart(rct)`, `checkDuration(rct)`, `checkCapacity(rct)` are called as soon as the start, duration or capacity variable of the resource constraint rct has changed[4]. The function `checkContribution(rct)` is attached to a $whenContribution$ event on the MPG ; it is called as soon as the contribution of a resource constraint has changed on the MPG . The function `checkDirectSuccessor(rct)` is attached to a $whenDirectSuccessor$ event on the MPG ; it is called as soon as new direct successors of rct have appeared on the MPG .

```
1.  checkContributionRelation(rct) {
2.    if C(rct)={1} {
```

[3]This semantics is the one already used by the disjunctive constraint to propagate transition times.

[4]The algorithm of the function `checkEnd(rct)` is symmetrical to the one of `checkStart(rct)`, that's why we don't describe it.

```
3.      setIsTrue(rct);
3.      setDurationMin(rct,1);
4.      setCapacityMin(rct,1);
5.    } else if C(rct)={0} {
6.      if dmin(rct)>0 and cmin(rct)>0 and not isToBeFalse(rct)
7.        setIsFalse(rct);
8.      if isToBeTrue(rct) and cmin(rct)>0 and dmin(rct)=0
9.        setDurationMax(rct,0);
10.     if isToBeTrue(rct) and dmin(rct)>0 and cmin(rct)=0
11.       setCapacityMax(rct,0);
12.     if isToBeTrue(rct) and dmin(rct)>0 and cmin(rct)>0
13.       fail();
14.   } else {
15.     if isToBeTrue(rct) and dmin(rct)>0 and cmin(rct)>0
16.       setToContribute(rct);
17      else if isToBeFalse(rct) or dmax(rct)=0 or cmax(rct)=0
18.       setToNotContribute(rct);
19.   }
20. }
```

We give here the algorithm of `checkContributionRelation(rct)` in the case rct is a capacity resource with a time extent $FromStartToEnd$. This function propagates the relations linking the contribution of the resource constraint to the duration and capacity variables as defined in section 3.2.2.

```
1.  checkSuccessorRelation(rct1, rct2) {
2.    if C(rct1)={1} and C(rct2)={1} {
3.      setEndMax(rct1, smax(rct2) - ttEndStart(rct1,rct2));
4.      setStartMin(rct2, emin(rct1) + ttEndStart(rct1,rct2));
5.    } else if (emin(rct1) + ttEndStart(rct1,rct2) > smax(rct2))
            or (smax(rct2) - ttEndStart(rct1,rct2) < emin(rct1)) {
6.      if C(rct2)={1}
7.        setToNotContribute(rct1);
8.      if C(rct1)={1}
9.        setToNotContribute(rct2);
10.   }
11. }
```

The function `checkSuccessorRelation(rct1, rct2)` propagates the temporal relation defined in section 3.2.3 between two resource constraints $rct1$ and $rct2$ such that $rct2$ is a successor of $rct1$.

```
1.  checkStart(rct) {
2.    if transitionTimeSatisfyTriangularInequality()
3.      rctPred = DPred(rct);
4.    else
5.      rctPred = Pred(rct);
```

```
6.    for (rctP in rctPred)
7.      checkSuccessorRelation(rctP, rct);
8.  }
```

It is interesting to notice that when the transition times satisfy the triangular inequality, it is sufficient to propagate the relation seen in section 3.2.3 on the direct successors of rct_1 instead of all its successors. This particularity is used to optimize the performances of the propagation at lines 2-5.

```
1.  checkDuration(rct) {
2.    checkContributionRelation(rct);
3.  }
```

When the duration variable of a resource constraint changes, it is sufficient to propagate the relations linking the contribution of the resource constraint to the duration and capacity variables (see section 3.2.2).

```
1.  checkCapacity(rct) {
2.    checkContributionRelation(rct);
3.  }
```

In the same way, when the capacity variable of a resource constraint changes, it is sufficient to propagate the relations linking the contribution of the resource constraint to the duration and capacity variables (see section 3.2.2).

```
1.  checkContribution(rct) {
2.    checkContributionRelation(rct);
3.    if C(rct)={1} {
4.      if transitionTimeSatisfyTriangularInequality() {
5.        rctPred = DPred(rct);
6.        rctSucc = DSucc(rct);
7.      } else {
8.        rctPred = Pred(rct);
9.        rctSucc = Succ(rct);
10.     }
11.     for (rctS in rctSucc)
12.       checkSuccessorRelation(rct,rctS);
13.     for (rctP in rctPred)
14.       checkSuccessorRelation(rctP,rct);
15.   }
16. }
```

When the contribution of a resource constraint changes on the MPG , the function `checkContribution(rct)` first propagate the relations linking the contribution of the resource constraint to the duration and capacity variables as defined in section 3.2.2 (line 2). Then, it propagates the successor relations as defined by relation of section 3.2.3 (lines 11-15).

```
1.  checkDirectSuccessor(rct) {
2.    for (rctNDS in DeltaDSucc(rct)) {
```

```
3.      checkSuccessorRelation(rct, rctNDS);
4.      if not transitionTimeSatisfyTriangularInequality() {
5.        for (rctNDSS in Succ(rctNDS))
6.          checkSuccessorRelation(rct, rctNDSS);
7.        for (rctP in Pred(rct)) {
8.          checkSuccessorRelation(rctP, rct);
9.          for (rctNDSS in Succ(rctNDS))
10.           checkSuccessorRelation(rctP, rctNDSS);
11.       }
12.     }
13.   }
14. }
```

The function `checkDirectSuccessor(rct)` propagates the new direct successor relations as defined by relation of section 3.2.3. Notice that when the transition time does not satisfy the triangular inequality, if (rct_1, rct_2) is a new direct successor relation, it is necessary to propagate the relation between all the couples $(rctP_1, rctS_2)$ where $rctP_1$ is a predecessor of rct_1 and $rctS_2$ a successor of $rct2$. This is done in lines 4-13 of the algorithm.

3.2.5 Comparison with Existing Scheduler Constraints

Successor relations. There are two differences between a successor relation between two resource constraints (rct_1,rct_2) on the MPG and a precedence constraint ($startsAfterEnd$) between their activities. The first difference is a semantical difference: the precedence constraint will be propagated independently from the fact that the resource constraints effectively require the resource or not whereas the successor relation on the MPG will be propagated only when the resource constraints effectively requires the resource. As such, from this point of view, successor relations are more rich than precedence constraints. Now, even in the case both resource constraint effectively affect the availability of the resource, the second difference is that successor relations propagate more because they take transition times into account. This can be easily illustrated by the example below.

Example 18 *Suppose a resource with three resource constraints $rct_0(type = 0)$, $rct1(type = 1)$ and $rct_2(type = 2)$. The duration of all the activities is 10 and the transition times are given by the following matrix on types:*

$$TT = \begin{pmatrix} 0 & 20 & 60 \\ 20 & 0 & 20 \\ 60 & 20 & 0 \end{pmatrix}$$

Suppose that two precedence constraints $startsAfterEnd$ are posted: rct_1 starts after ends rct_0 and rct_2 starts after ends rct_1. If the horizon of the schedule is large, it will not allow the disjunctive constraint or the type timetable to propagate the transition times along the precedence constraints and, the minimal start time of rct_2 will be 20. Now, if a MPG constraint is used and if the precedences are posted as successor relations on the MPG , the transition times will be propagated whatever the horizon of the schedule and the minimal start time of rct_2 will be 70[5].

[5]The corresponding code using ILOG Scheduler 4.4 can be found in `/nfs/schedule/techreports/OIR-1999-1/src/ttimeExtraProp.cpp`

This extra-propagation may turn out to be very helpful. For example, we compared the two approaches when optimizing the $trolley2$ example in [4], Chapter 19. This example uses a state resource ($trolley$) with transition times and precedence constraints. The results are given on the table below. The running time corresponds to the CPU time on a sparc Ultra-5_10.

	With precedence	With MPG
Number of fails	32899	9001
Memory used (bytes)	122068	238648
Running time (s)	51.6	10.17

Next relations. On a unary resource, the semantics of a next relation between two resource constraints (rct_i, rct_j) is the same as posting the following constraint between the next and previous variables of the resource constraints[6]:

$$(next(rct_i) \geq 0) \wedge (prev(rct_j) \geq 0) \Rightarrow next(rct_i) = j$$

In particular, when both resource constraints contribute, the semantics is the same as instantiating the next variable of rct_i and the previous variable of rct_j. Thus, the MPG constraint can also be seen as an efficient "light" version of the sequence constraint. For example, suppose you want to solve a problem on unary resources with the additional constraints that some couples of activities are constrained to execute one next to the other on their resource, using the MPG alone instead of the sequence constraint may dramatically improve the performances. We can illustrate it on a scheduling problem with a single unary resource and n activities that require this resources[7]. We suppose that the n activities are decomposed into $\sqrt{n}$ sequences of $\sqrt{n}$ activities. The curves on figure 3.2 compares the CPU time taken for solving the problem with a sequence constraint (the chains of activities are build by instantiating the next variable of the activities) and with a MPG constraint only (the chains are build by inserting next relations on the MPG) for different values of n. The times are given in s on a $HP - UX9000/780$ station.

[6]For simplicity, we suppose that i and j are the indexes of rct_i and rct_j on the sequence constraint.

[7]The corresponding code using ILOG Scheduler 4.4 can be found in `/nfs/schedule/techreports/OIR-1999-1/src/lightSequence.cpp`

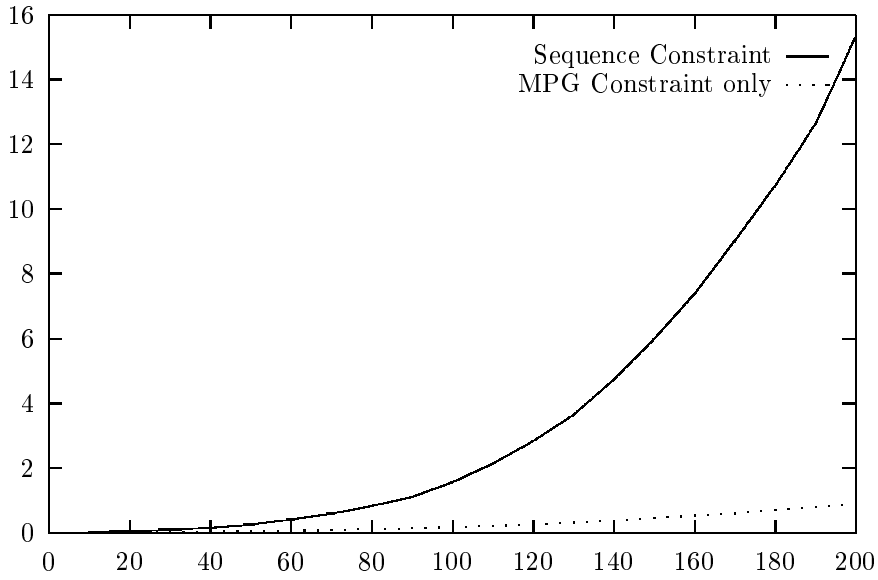

Figure 3.2: The MPG Constraint as a Light Sequence Constraint

Precedence Constraint	Conditions	Successor Relation
$a_1.startsAfterStart(a_2, d)$	$(d \geq dmax_2) \vee$ $(I \wedge (d > -dmin_1))$	$rct_2 \rightarrow rct_1$
$a_1.startsAfterEnd(a_2, d)$	$(d \geq 0) \vee$ $(I \wedge (d > -dmin_1 - dmin_2))$	$rct_2 \rightarrow rct_1$
$a_1.endsAfterStart(a_2, d)$	$(d \geq dmax_1 + dmax_2) \vee$ $(I \wedge (d > 0))$	$rct_2 \rightarrow rct_1$
$a_1.endsAfterEnd(a_2, d)$	$(d \geq dmax_1) \vee$ $(I \wedge (d > -dmin_2))$	$rct_2 \rightarrow rct_1$
$a_1.startsAtStart(a_2, d)$	$(d \geq dmax_2) \vee$ $(I \wedge (d > -dmin_1))$	$rct_2 \rightarrow rct_1$
$a_1.startsAtStart(a_2, d)$	$(d \leq -dmax_1) \vee$ $(I \wedge (d < dmin_2))$	$rct_1 \rightarrow rct_2$
$a_1.startsAtEnd(a_2, d)$	$(d \geq 0) \vee$ $(I \wedge (d > -dmin_1 - dmin_2))$	$rct_2 \rightarrow rct_1$
$a_1.startsAtEnd(a_2, d)$	$(d \leq -dmax_1 - dmax_2) \vee$ $(I \wedge (d < 0))$	$rct_1 \rightarrow rct_2$
$a_1.endsAtStart(a_2, d)$	$(d \geq dmax_1 + dmax_2) \vee$ $(I \wedge (d > 0))$	$rct_2 \rightarrow rct_1$
$a_1.endsAtStart(a_2, d)$	$(d \leq 0) \vee$ $(I \wedge (d < dmin_1 + dmin_2))$	$rct_1 \rightarrow rct_2$
$a_1.endsAtEnd(a_2, d)$	$(d \geq dmax_1) \vee$ $(I \wedge (d > -dmin_2))$	$rct_2 \rightarrow rct_1$
$a_1.endsAtEnd(a_2, d)$	$(d \leq -dmax_2) \vee$ $(I \wedge (d < dmin_1))$	$rct_1 \rightarrow rct_2$

Table 3.1: Correspondence between Precedence Constraints and Successor Relations

On non-unary resources, the semantics of a next relation between a couple of resource constraints (rct_i, rct_j) is less intuitive to describe. It states that whenever both resource constraints rct_i and rct_j effectively affects the availability of the resource, there cannot be another resource constraint rct_k that effectively affects the availability of the resource and that either (1) starts between the end of rct_i and the end of rct_j or (2) ends between the start of rct_i and the start of rct_j.

3.3 Integration with Existing Scheduler Constraints

3.3.1 Precedence Constraints

When a MPG constraint is created on a resource, some precedence constraints can automatically be inserted as successor edges on the MPG . This is the case for the precedence constraints on table 3.1 when they constrain two activities that require the same resource. The condition I in table 3.1 is true when and only when the two resource constraints cannot overlap when they both contribute; this is for example systematically the case for a unary resource.

Such an automatic insertion on new successor edges in the MPG by precedence constraints can easily be implemented by associating, for each precedence constraint, some demons attached to

Precedence Constraint	Conditions	Successor Relation
$a_1.startsAfterEnd(a_2, d)$	$(d \geq 0)$	$rct_2 \rightarrow rct_1$
$a_1.startsAtEnd(a_2, d)$	$(d \geq 0)$	$rct_2 \rightarrow rct_1$
$a_1.endsAtStart(a_2, d)$	$(d \leq 0)$	$rct_1 \rightarrow rct_2$

Table 3.2: Restricted Correspondence as Implemented in Scheduler 4.4

the modification of the duration variable of activities. In Scheduler 4.4, only a restricted version of this table of correspondence has been implemented; it corresponds to table 3.2. Notice anyway that the most common precedence constraints are covered (e.g. $startsAfterEnd$ with positive delay). For example, when we shown the extra-propagation due to the MPG constraint in the section 3.2.5, the only difference between the two approaches with or without MPG was an additional line `m.add(trolley.makePrecedenceGraphConstraint());` in the version with MPG , the precedence constraints being automatically taken into account by the MPG .

3.3.2 Disjunctive Constraint

When no MPG constraint is defined on a resource, the Disjunctive Constraint of ILOG Scheduler is in charge of discovering and propagating precedence relations between resource constraints that cannot overlap because of resource capacity usage. Basically, the following propagation rule is applied for all the couple (rct_i, rct_j):

1. $I(rct_i, rct_j)$
2. $\wedge$ $Contribute(rct_i)$
3. $\wedge$ $Contribute(rct_j)$
4. $\wedge$ $(emin(rct_j) + ttEndStart(rct_j, rct_i) > smax(rct_i))$
5. $\Rightarrow$ $smin(rct_j) = max(smin(rct_j), emin(rct_i) + ttEndStart(rct_i, rct_j))$
6. $\wedge$ $emax(rct_i) = min(emax(rct_i), smax(rct_j) - ttEndStart(rct_i, rct_j))$

Where $I(rct_i, rct_j)$ is true when and only when the two resource constraints cannot overlap when they both contribute (this is for example systematically the case for a unary resource). And where $Contribute(rct)$ is true is and only if rct satisfies the definitions in section 3.2.2.

When a MPG constraint is defined on the resource, we see that the relation between the contribution status of the resource constraints (points 2 and 3) and the temporal constraints (points 5 and 6) can be directly propagated thanks to a successor relation $rct_i \rightarrow rct_j$ on the MPG . Thus, the disjunctive constraint only needs to ensure that:

1. $I(rct_i, rct_j)$
2. $\wedge$ $(emin(rct_j) + ttEndStart(rct_j, rct_i) > smax(rct_i))$
3. $\Rightarrow$ $rct_j \in Succ(rct_i)$

Furthermore, it only needs to analyze those couples (rct_i, rct_j) that are still unranked on the MPG . When a MPG constraint is defined on the resource, the disjunctive constraint is triggered as soon as the start or end variable of a resource constraint rct is modified. Then, it scans all the resource constraints $rct_U \in Urkd(rct)$[8] and try to apply the propagation rule above to discover new

[8]Remember from Chapter 2 that the complexity of scanning $Urkd(rct)$ is strictly equal to the size of this set.

successor relations $rct_U \rightarrow rct$ or $rct \rightarrow rct_U$ on the MPG . When there is no MPG constraint, the Disjunctive Constraint needs to systematically scan all the resource constraints instead of only the ones unranked with respect to the current resource constraint. This is why, although the MPG constraint does not allow the Disjunctive Constraint to propagate more, it can help it to propagate quicker. This performance improvement becomes very sensible when the density of successor relations in the MPG is high so that the sets $Urkd(rct)$ are small. This phenomenon is illustrated below.

Suppose a very simple scheduling problem with n activities with the same duration on a unique unary resource. The problem consists in searching for the 1000 first solutions with the ILOG scheduler goal $IlcRank$[9]. The curves on figure 3.3 show how the time taken to solve this problem scales with n when only a disjunctive constraint is posted and when the disjunctive constraint is associated with a MPG constraint[10]. The y-axis shows the CPU time in s on a $HP - UX9000/780$ station.

Anyway, it would be a too hative conclusion to say that the introduction of the MPG Constraint always improve the time performance of the Disjunctive Constraint. With the version implemented in Scheduler 4.4, we found that, for small jobshop problems (10-20 activities on each resource), the introduction of the MPG constraint, in cases where this constraint is not exploited (no extra-propagation switched on, no search based on the MPG), could actually slow down the time performances until about 20%.

[9]The horizon is supposed to be large enough to ensure that a solution can be found. It is clear that the problem has exactly $n!$ solutions.

[10]The corresponding code using ILOG Scheduler 4.4 can be found in `/nfs/schedule/techreports/OIR-1999-1/src/disjunctiveImprove.cpp`

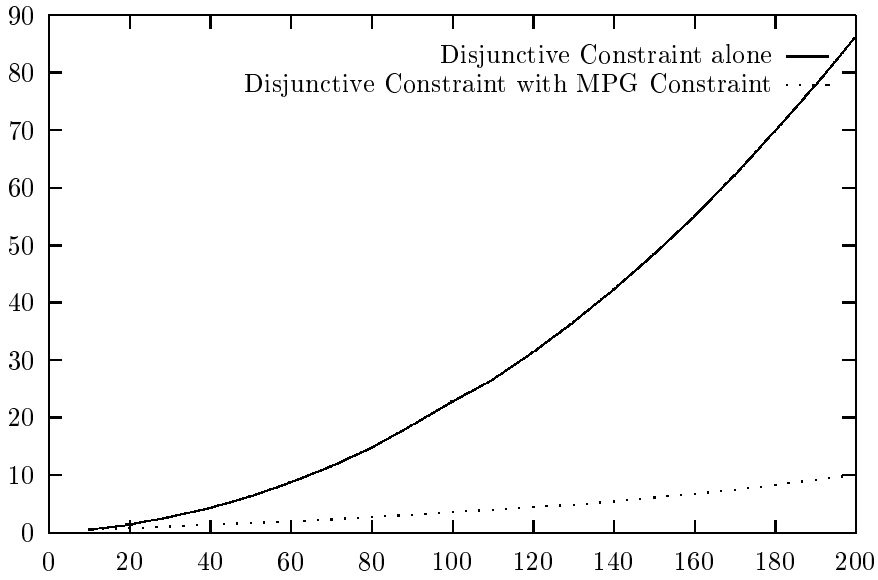

Figure 3.3: Influence of MPG Constraint on Disjunctive Constraint

3.3.3 Sequence Constraint

A MPG Constraint is automatically created when a sequence constraint is defined on a resource. We chosed this approach for the following reasons:

1. The MPG Constraint allows to systematically improve the performances of the Sequence Constraint;
2. The association MPG Constraint + Sequence constraint allows to propagate more than the Sequence Constraint alone;
3. The extra-memory consumption due to the MPG Constraint is, in general, small compared to the memory already used by the Sequence Constraint (about 20% more).
4. The automatical creation of the MPG constraint when the Sequence Constraint is created allows to simplify the implementation as we don't need to handle the case when the Sequence Constraint have no MPG to rely on.

On the Sequence Contraint of ILOG Scheduler 4.4, most of the reasoning on the relative positions of activities on the resource is handled by the MPG . The role of the Sequence Constraint is to maintain the coherence between the previous/next variables of the resource constraints, the cost variables and the MPG . More precisely, the MPG allows to reduce the domain of the previous/next variables of resource constraints as the following relations must hold:

$$Next_{Sequence}(rct) \in PNext_{MPG}(rct)$$

$$Previous_{Sequence}(rct) \in PPrev_{MPG}(rct)$$

$$Contr_{MPG}(rct) = \{0\} \Rightarrow (Next_{Sequence}(rct) = -1) \wedge (Previous_{Sequence}(rct) = -1)$$

Thus, some demons that propagate these relations are attached to the events $whenPossibleNext$, $whenPossiblePrevious$ and $whenContribution$ of the MPG . Furthermore, the Sequence Constraint updates the MPG when a given resource constraint has its previous or next variable instantiated either to -1 (in that case, the resource constraint does not contribute), or to the index of another resource constraint (in that case, a next relation is inserted on the MPG). More formally, the relations under are propagated:

$$Next_{Sequence}(rct_i) = rct_j \Rightarrow rct_j \in Next_{MPG}(rct_i)$$

$$Previous_{Sequence}(rct_j) = rct_i \Rightarrow rct_j \in Next_{MPG}(rct_i)$$

$$(Next_{Sequence}(rct_i) = -1) \vee (Previous_{Sequence}(rct_i) = -1) \Rightarrow Contr_{MPG}(rct_i) = \{0\}$$

It's easy to see that the use of MPG whithin the Sequence Constraint allows to improve its propagation. Let's illustrate this with a short example.

Example 19 *Consider a schedule with an horizon of 100 and a unary resource with 3 resource constraints rct_1, rct_2 and rct_3 such that rct_1 is constrained to execute before rct_2 and rct_2 is constrained to execute before rct_3 (the duration of the activities is for example 10). Thanks to the MPG constraint, the Sequence Constraint will discover that the resource is completly sequenced ($Next_{Sequence}(rct_1) = rct_2$, $Next_{Sequence}(rct_2) = rct_3$) whereas, if there is no MPG Constraint, the Sequence Constraint does not reduce the initial domain of the previous/next variables*[11].

[11]The corresponding code using ILOG Scheduler 4.4 can be found in `/nfs/schedule/techreports/OIR-1999-1/src/sequenceExtraProp.cpp`

We have studied the performance improvements on the sequence constraint due to the usage of MPG [12]. We have seen in section 3.3.2, that in most cases, the MPG constraint improves the time performances of the disjunctive constraint. As a disjunctive constraint is automatically created when a sequence constraint is used on a unary resource, we have tried to put ourselve in the worst case where the MPG constraint does not improve the performances of the disjunctive constraint and when there is no extra-propagation of the Sequence Constraint due to the usage of MPG . This is the case for the jobshop problems benchmarks. Figure 3.4 compares the performances of the sequence of Scheduler 4.3 with the one of Scheduler 4.4 (with MPG) when we try to optimize some job-shop problems with the ILOG Scheduler goal `IlcSequence` and when the number of failures is limited to 2000. In order not to interfere with other improvements in Scheduler 4.4, the only constraints used for these problems were the disjunctive and the sequence constraint (and thus, the MPG constraint in Scheduler 4.4). Of course, this is not an efficient way for optimizing job-shop problems; just remember that our purpose here is only to compare the performances between the two versions of the sequence. The x-axis on the curve represents the 48 jobshop problems[13], sorted in such way that the curve $Sequence\ Scheduler\ 4.3$ is increasing. The curve $Sequence\ Scheduler\ 4.3$ represents the CPU time in s to solve the corresponding jobshop problem using ILOG Scheduler 4.3 whereas the curve $Sequence\ Scheduler\ 4.4$ represents the CPU time with the sequence of Scheduler 4.4 for the same problem. We have measured that in average the MPG constraint allows an improvement of 40% of the sequence constraint on these problems.

[12]The corresponding code using ILOG Scheduler 4.4 can be found in `/nfs/schedule/techreports/OIR-1999-1/src/sequenceImprove.cpp`

[13]Here is the exhaustive list: abz5-9, ft06, ft10, ft20, orb1-10, la01-30.

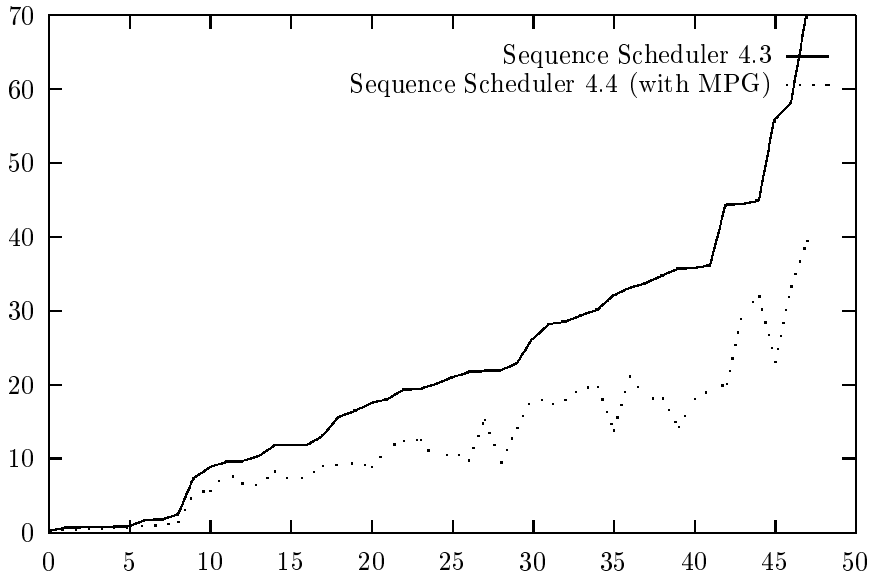

Figure 3.4: Influence of MPG Constraint on Sequence Constraint

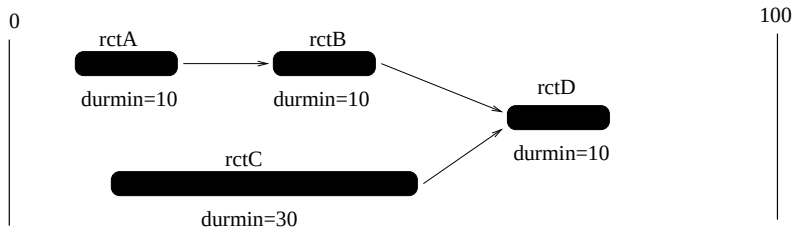

Figure 3.5: Propagation on a Unary Resource Using Modal Precedence Graphs

3.4 Some New Constraints Based on Modal Precedence Graphs

3.4.1 Unary Resources

We describe in this section a new constraint on unary resources based on MPG [14]. Given a unary resource R that is associated a MPG G, this constraint reduces the domain of the start and end variables of the activities that requires R by analyzing the successor relations on G. The principle of the constraint is a direct consequence of the following property:

Property 7 *Let RCT_+ denotes the set of contributor resource constraints on a unary resource R. Any consistent schedule must satisfy the following properties:*

- *(Forward)* $\forall rct \in RCT_+, \forall P \subset Pred(rct)$, *let* $rct_P \in P \; such \; that \; \forall rct_i \in P, smin(rct_P) \leq smin(rct_i)$ *then:*

$$smin(rct_P) + \sum_{rct_i \in P} dmin(rct_i) \leq smin(rct)$$

- *(Backward)* $\forall rct \in RCT_+, \forall S \subset Succ(rct)$, *let* $rct_S \in S \; such \; that \; \forall rct_i \in S, emax(rct_S) \geq emax(rct_i)$ *then:*

$$emax(rct) \leq emax(rct_S) - \sum_{rct_i \in S} dmin(rct_i)$$

Example 20 *Suppose four resource constraints rct_A, rct_B, rct_C and rct_D on a unary resource as described in figure 3.5 on a schedule with an horizon equal to 100. If we apply the property (Forward) to the resource constraint rct_D with $P = \{rct_A, rct_B, rct_C\}$ we will obtain a new minimal start time of 50 for rct_D. The disjunctive and the edge-finder constraints alone cannot make such a propagation.*

Propagation Algorithm. The algorithm works in 2 steps, first it ensures property (Forward) then property (Backward). Let's focus on property (Forward) as property (Backward) is completely symmetrical. The algorithm first sort the resource constraints of RCT_+ by increasing earliest start time of activities. Each sorted resource constraint is stored in an array (line 2). The index of the resource constraint in the array corresponds to its rank in the sort. Then, it initialize an array of integers ($sumDurPred$). $sumDurPred[i]$ is initialized with the sum of the minimal duration of the predecessors of the rct of rank i (lines 4-8). The resource constraints rct_i are then scanned by increasing

[14] This functionality has been implemented in ILOG Scheduler 4.4.

order of earliest start time (lines 9-14). For a resource constraint rct_i, the algorithm looks all the resource constraints rct_j such that rct_i is a predecessor of rct_j (line 10). In such way, with the loop on rct_i, the predecessors of rct_j will be scanned by increasing earliest start time. The cumulated sum of the duration min of the predecessors of rct_j whose earliest start time is greater or equal to the one of rct_i is maintained incrementally. This cumulated time is used to compute the new earliest start time of rct_j (line 12). Line 13 updates this cumulated time. Notice that the algorithm needs a direct access to the index of a resource constraint in the array (line 11).

```
1.  propagateForward() {
2.    rctSMin = qsort(RCT+, increasingEarliestStartTime);
3.    n = size(RCT+);
4.    for (i = 1 to n) {
5.      sumDurPred[i] = 0;
6.      for (rct in Pred(rctSMin[i]))
7.        sumDurPred[i] = sumDurPred[i] + dmin(rct);
8.    }
9.    for (i = 1 to n)
10.     for (rctj in Succ(rctSMin[i])) {
11.       j = getIndexInArrayRctSmin(rctj);
12.       setStartMin(rctSMin[j], smin(rctSMin[i]) + sumDurPred[j]);
13.       sumDurPred[j] = sumDurPred[j] - dur(rctSMin[i]);
14.     }
15. }
```

Complexity. Suppose that s is the number of successor edges on the closure of the MPG , the complexity of the first loop (lines 4-8) is in $O(s)$ as well as the complexity of the second loop (lines 9-14). As the sort is in $O(n.log(n))$, it leads to a global complexity in $O(n.log(n) + s)$ that is, $O(n^2)$ in worst case. Notice that a lighter version of this constraint has also been implemented. This version only analyses the direct predecessors/successors of a given resource constraint instead of all the predecessors/successors. In that case, if ds is the number of direct edges on the graph, it is easy to show that the propagation of the constraint is in $O(n.log(n) + ds)$ in worst case. Of course, the propagation is in that case weaker; for example, on figure 3.5, it would find a new minimal start time of 40 for rct_D.

3.4.2 Discrete Resources

The constraint above can easily be generalized to discrete resources. In that case, the idea is to push a resource constraint rct so that there is enough energy before rct to execute all its predecessors. This is formalized in the following property:

Property 8 *Let RCT_+ denotes the set of contributor resource constraints on a discrete resource R. C denotes the maximal capacity of R. Any consistent schedule must satisfy the following properties:*

- *(Forward)* $\forall rct \in RCT_+, \forall P \subset Pred(rct)$,
  let $rct_P \in P$ *such that* $\forall rct_i \in P, smin(rct_P) \leq smin(rct_i)$ *then:*

$$smin(rct_P) + \sum_{rct_i \in P} \frac{cmin(rct_i) * dmin(rct_i)}{C} \leq smin(rct)$$

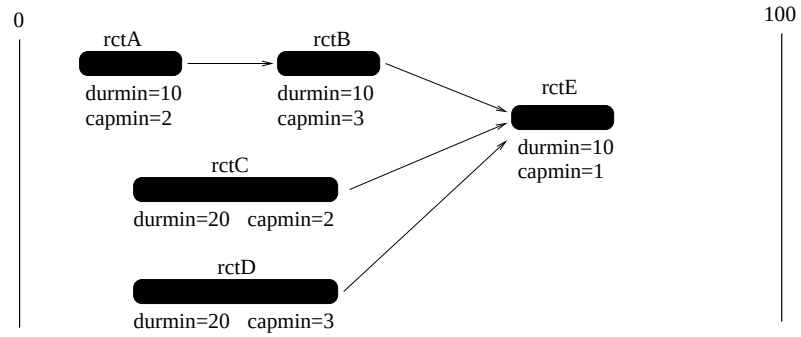

Figure 3.6: Propagation on a Discrete Resource Using Modal Precedence Graphs

- *(Backward)* $\forall rct \in RCT_+, \forall S \subset Succ(rct)$,
  let $rct_S \in S$ *such that* $\forall rct_i \in S, emax(rct_S) \geq emax(rct_i)$ *then:*

$$emax(rct) \leq emax(rct_S) - \sum_{rct_i \in S} \frac{cmin(rct_i) * dmin(rct_i)}{C}$$

The same algorithm as for unary resources can be implemented[15]; we just need to replace the occurrences of $dmin(rct_i)$ by $cmin(rct_i) * dmin(rct_i)/C$. Here also, a lighter version of the algorithm can be implemented that only works on direct predecessors/successors instead of predecessor/successors. The complexity of the algorithms is the same as for unary resources.

Example 21 *Suppose five resource constraints* rct_A, rct_B, rct_C, rct_D *and* rct_E *on a discrete resource of capacity* 5 *as described in figure 3.6 on a schedule with an horizon equal to 100. If we apply the property (Forward) to the resource constraint* rct_E *with* $P = \{rct_A, rct_B, rct_C, rct_D\}$ *we will obtain a new minimal start time of* 30 *for* rct_D*. As for unary resources, the disjunctive and the edge-finder constraints alone cannot make such a propagation.*

Remark. A similar constraint could also be implemented on discrete energy resources. Notice that for discrete energy the problem is slighly more complex because of the rounding on timesteps. Anyway, as discrete energy resources are mostly used as a relaxation of discrete resource for very large problems, the practical interest of such a constraint using MPG is more limited.

3.4.3 Reservoirs

Notations. Let R be a reservoir and RCT all the resource constraints on R. We will denote $Cons \subset RCT$ the set of consumption resource constraints and $Prod \subset RCT$ the set of production resource constraints. We suppose in this section that reservoir consumption resource constraints consume the resource at their start time and that production resource constraints produce the resource at their end time. If rct is a reservoir consumption resource constraint, we will denote $cmin(rct)$ and $cmax(rct)$ respectively the minimal and maximal capacity consumed by rct. If rct is a reservoir production resource constraint, we will denote $pmin(rct)$ and $pmax(rct)$ respectively the minimal and maximal

[15]This functionality has been implemented in ILOG Scheduler after the release of ILOG Scheduler 4.4.

capacity produced by rct. For homogeneity reasons, when rct is a reservoir consumption resource constraint, we will consider $pmin(rct) = pmax(rct) = 0$ and when rct is a reservoir production resource constraint, we will consider $cmin(rct) = cmax(rct) = 0$. We will denote L_{init} the initial level of the reservoir, L_{min} its minimal level and L_{max} its maximal level. For a given resource constraint rct, let:

$$CBmin(rct) = \sum_{rct_C \in Pred(rct)} cmin(rct_C)$$

$$CBmax(rct) = \sum_{rct_C \notin Succ(rct)} cmax(rct_C)$$

$$PBmin(rct) = \sum_{rct_P \in Pred(rct)} pmin(rct_P)$$

$$PBmax(rct) = \sum_{rct_P \notin Succ(rct)} pmax(rct_P)$$

If the reservoir is closed, it is clear that: $CBmin(rct)$ is a lower bound on the quantity of resource that could be consumed before the start time of rct; $CBmax(rct)$ is an upper bound on the quantity of resource that could be consumed before the start time of rct; $PBmin(rct)$ is a lower bound on the quantity of resource that could be produced before the start time of rct and $PBmax(rct)$ is an upper bound on the quantity of resource that could be produced before the start time of rct. The following properties are a consequence of the fact that during the execution of a resource constraint, the availability of the reservoir must be at least its minimal capacity and at most its maximal capacity.

Property 9 *If the reservoir is closed, any consistent schedule must satisfy the following properties for all rct in RCT:*

$$L_{init} + PBmax(rct) - CBmin(rct) - cmin(rct) \geq L_{min}$$

$$L_{init} + PBmin(rct) - CBmax(rct) + pmin(rct) \leq L_{max}$$

These properties can directly be used in a constraint that prune the search space. But we can still do better; the quantity $PBmax(rct)$ that is an upper bound on the quantity of resource that could be produced before rct can be seen as the sum of $PBBmax(rct)$ an upper bound on the quantity of resource that could be produced by the resource constraints currently before rct and $PUBmax(rct)$ an upper bound on the quantity of resource that could be produced by the resource constraints still unranked with respect to rct. Thus:

$$PBBmax(rct) = \sum_{rct_P \in Pred(rct)} pmax(rct_P)$$

$$PUBmax(rct) = \sum_{rct_P \in Urkd(rct)} pmax(rct_P)$$

In the same way, the quantity $CBmax(rct)$ that is an upper bound on the quantity of resource that could be consumed before rct can be seen as the sum of $CBBmax(rct)$ an upper bound on

the quantity of resource that could be consumed by the resource constraints currently before rct and $CUBmax(rct)$ an upper bound on the quantity of resource that could be consumed by the resource constraints still unranked with respect to rct. Thus:

$$CBBmax(rct) = \sum_{rct_C \in Pred(rct)} cmax(rct_C)$$

$$CUBmax(rct) = \sum_{rct_C \in Urkd(rct)} cmax(rct_C)$$

Now, let's consider two the following quantities:

$$LackB(rct) = L_{min} + cmin(rct) + CBmin(rct) - PBBmax(rct) - L_{init}$$

$$ExcessB(rct) = L_{init} + PBmin(rct) - CBBmax(rct) + pmin(rct) - L_{max}$$

When $LackB(rct) > 0$, it means that it is sure the current predecessors of rct does not produce enough resource to ensure that at the start of rct the level will be at least $L_{min} + cmin(rct)$. Thus, several production resource constraints that are currently unranked with respect to rct will have to be ranked before rct. Although we don't know exactly which ones, we still can improve the minimal start time of rct. Indeed, we know that these new predecessors will have to produce at least $LackB(rct)$ units of reservoir thus, in the best case, rct must be pushed in the future till the end time of the earliest subset of producers unranked with respect to rct that could produce $LackB(rct)$. Symmetrically, when $ExcessB(rct) > 0$, it means that it is sure the current predecessors of rct produce too much resource to ensure that at the start of rct the level will be less than $L_{max} - pmin(rct)$. Thus, several consumption resource constraints that are currently unranked with respect to rct will have to be ranked before rct. Although we don't know exactly which ones, we still can improve the minimal start time of rct. We know that these new predecessors will have to consume at least $ExcessB(rct)$ units of reservoir thus, in the best case, rct must be pushed in the future till the start time of the earliest subset of consumers unranked with respect to rct that could consume $ExcessB(rct)$. This is formalized in the property under (notice that items 1 and 2 are just a reformulation of property 9).

Property 10 *If the reservoir is closed, any consistent schedule must satisfy the following properties for all rct in RCT:*

1. $LackB(rct) \leq PUBmax(rct)$
2. $ExcessB(rct) \leq CUBmax(rct)$
3. *If* $0 < LackB(rct) \leq PUBmax(rct)$,
   Let $PUListEMIN(rct) = (rct_0, ..., rct_k)$ *be the list of the resource productors unranked with respect to* rct *and sorted by increasing minimal end time.*
   Let $s \in [0, k]$ *be the smallest index such that:* $\sum_{i=0}^{s} pmax(rct_i) \geq LackB(rct)$
   Then $smin(rct) \geq emin(rct_s)$
4. *If* $0 < ExcessB(rct) \leq CUBmax(rct)$,
   Let $CUListSMIN(rct) = (rct_0, ..., rct_k)$ *be the list of the resource consumers unranked with respect to* rct *and sorted by increasing minimal start time.*
   Let $s \in [0, k]$ *be the smallest index such that:* $\sum_{i=0}^{s} cmax(rct_i) \geq ExcessB(rct)$
   Then if $rct \in Cons$, $smin(rct) \geq smin(rct_s)$*; else if* $rct \in Prod$, $emin(rct) \geq smin(rct_s)$.

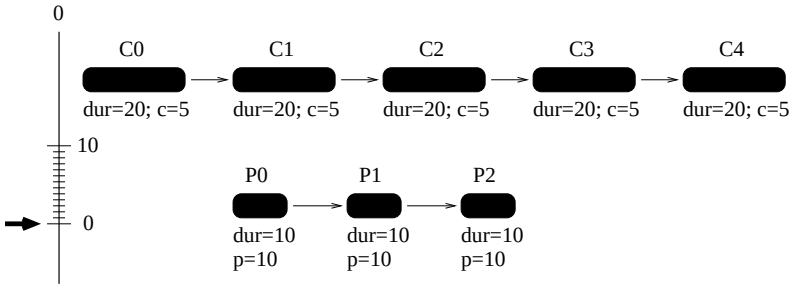

Figure 3.7: Propagation on a Reservoir Using Modal Precedence Graphs

Before we describe an efficient algorithm to perform this propagation, let's illustrate it on an example.

Example 22 *Consider a schedule with a large horizon (for example 1000), a reservoir of maximal level 10 and initial level 0. There is a sery of 5 consuming activities C_0...C_4, all of them have a duration of 20 and consume 5 units of the reservoir; they are constrained to precede one another. There are also a sery of 3 producing activities P_0, P_1, P_2, all of them have a duration of 10 and produce 10 units of reservoir. They are constrained to precede one another. The reservoir is supposed to be closed. This schedule is illustrated on figure 3.7. If we apply property 10 on P_2 for example, we have: $PBmin(P_2) = pmin(P_0) + pmin(P_1) = 20$ and $CBBmax(P_2) = 0$ thus $ExcessB(P_2) = 0 + 20 - 0 + 10 - 10 = 20$. We have also $CUBmax(P_2) = cmax(C_0) + cmax(C_1) + cmax(C_2) + cmax(C_3) + cmax(C_4) = 25$. So we see that $0 < ExcessB(P_2) \leq CUBmax(P_2)$. If we apply point 4 of the property, we see that $CUListSMin = (C_0, C_1, C_2, C_3, C_4)$, $s = 3$ and thus, we must have $emin(P_2) \geq smin(C_3) = 60$. When the horizon of the schedule is large, the reservoir timetable constraint does not make this propagation.*

Propagation Algorithm. The algorithm[16] can be decomposed in two parts: a first function (`propagateLack`) that propagates points 1 and 3 of property 10 and a function (`propagateExcess`) that propagates points 2 and 4. For an evident reason of symmetry, we will only describe here the function (`propagateLack`). This function first sorts the set of resource constraints by increasing earliest end time (line 2). Then, it computes for each resource constraints, the quantity $lackB(rct_i)$ (lines 4-8). After that, it will scan the resource constraints rct_j by increasing earliest end time (line 9) and, inside this loop, scan the resource constraints rct_i unranked with respect to rct_j (line 10). Seen from rct_i it is like if we were scanning the resource constraint rct_j unranked with respect to rct_i by increasing earliest end time that is, the list $PUListEMin(rct_i)$ of the property. For each rct_j, we decrease $lackB[rct_i]$ with the quantity $pmax(rct_j)$ until $lackB[rct_i] \leq 0$, then, we store the minimal end time of rct_j (line 14). Depending on the fact rct_i is a reservoir consumer or producer, this will be the new minimal start time or minimal end time of rct_i (lines 20-23). Notice also that, if after the loop at lines 9-16, there is still a resource constraint for which $lackB[rct_i] > 0$, then we can fail as point 1 of the property is not satisfied (lines 18-19).

```
1.  propagateLack() {
```

[16] This functionality has been implemented in ILOG Scheduler after the release of ILOG Scheduler 4.4.

```
2.    rctEMin = qsort(RCT, increasingEarliestEndTime);
3.    n = size(RCT);
4.    for (i = 1 to n) {
5.      lackB[i] = Lmin - Linit + cmin(rctEMin[i]);
6.      for (rct in Pred(rctEMin[i]))
7.        lackB[i] += cmin(rct) - pmax(rct);
8.    }
9.    for (j = 1 to n)
10.     for (rcti in Urkd(rctEMin[j])) {
11.       i = getIndexInArrayRctEmin(rcti);
12.       if (lackB[i] > 0) {
12.         lackB[i] -= pmax(rctEMin[j]);
13.         if (lackB[i] <= 0)
14.          storedEMin[i] = emin(rctEMin[j]);
15.       }
16.     }
17.   for (i = 1 to n) {
18.     if (lackB[i] > 0)
19.       fail;
20.     else if (rctEMin[i] in Cons)
21        setStartMin(rctEMin[i], storedEMin[i]);
22.     else
23.       setEndMin(rctEMin[i], storedEMin[i]);
24.   }
25. }
```

Complexity. Suppose that s and u are respectively the number of successor edges and the number of unranked pairs of resource constraints on the closure of the MPG . The complexity of the first loop (lines 4-8) is in $O(s)$. The complexity of the second loop (lines 9-16) is in $O(u)$. As the sort is in $O(n.log(n))$, it leads to a global complexity in $O(n.log(n) + s + u)$ that is, $O(n^2)$ in worst case.

3.5 Example of Goal using Modal Precedence Graphs

3.5.1 Least-Commitment Strategy

As an illustration of the use of MPG to define new search goals in ILOG Scheduler, we describe in this section a very simple goal called $LEASTCOM$ that ranks activities on unary resources[17]. The purpose of this goal is to reach very good first solutions. We will first describe the goal and then compare it on some jobshop problems with some state of the art greedy algorithms of the literature.

$LEASTCOM$ supposes that each unary resource of the schedule is associated a precedence graph constraint. The principle of this goal is to select, at each node of the search tree, a couple of resource constraints (rct_1, rct_2) on the same resource and to impose a precedence relation $rct_1 \rightarrow rct_2$ between them. In case of a failure, the alternative choice $rct_2 \rightarrow rct_1$ is tried[18]. Now, the critical point of the goal is the selection of the couple of resource constraints $(rct1, rct2)$.

[17]This goal is also described in the ILOG Scheduler User's Manual [6], Chapter 6.

[18]This is done with the predefined ILOG Scheduler goal $IlcTrySetSuccessor(rct_1, rct_2)$.

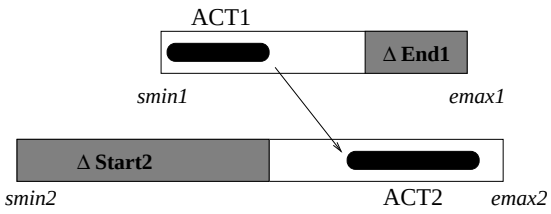

Figure 3.8: Impact of an Ordering Decision

If $\{rct_1, rct_2\}$ is a couple of unranked resource constraints, we define a criterion $impact(rct_1, rct_2)$ that approximates the impact on the schedule of the decision of ordering rct_1 before rct_2. The impact of a decision on the schedule is a quantity that is proportional to the domain reduction of the start and end variables of activities due to the propagation of the decision [7]. Suppose that act_1 and act_2 are the activities of rct_1 and rct_2, and that:

- $emin_1$ is the current minimal end time of act_1,
- $emax_1$ is the current maximal end time of act_1,
- $smin_2$ is the current minimal start time of act_2 and
- $smax_2$ is the current maximal start time of act_2.

When adding the precedence relation $rct_1 \rightarrow rct_2$, the reduction of the domain of the end time of act_1 will be at least $\Delta End_1 = emax_1 - min(emax_1, smax_2)$ and the reduction of the domain of the start time of act_2 will be at least $\Delta Start_2 = max(emin_1, smin_2) - smin_2$. See an illustration on figure 3.8. Thus, we estimate $impact(rct_1, rct_2) = \Delta End_1 + \Delta Start_2$

If we have the choice between ordering rct_1 before rct_2 or rct_2 before rct_1, we will always choose the order of minimal impact. Indeed, in such way, the domain reduction for the start and end variables of activities will be smaller and thus:

- we can hope to stay as close as possible to the initial domain of these variables and in such way, obtain a small makespan
- it will leave as much room as possible for future decisions.

We know that in all the solutions, all the couples $\{rct_1, rct_2\}$ will have to be ranked. As our goal is to minimize the impact of all the decisions we take until a solution is found, a good strategy is to select in priority those potential conflicts $\{rct_1, rct_2\}$ for which the choice of the decision between $rct_1 \rightarrow rct_2$ and $rct_2 \rightarrow rct_1$ is the most evident that is, the potential conflicts that maximize $|impact(rct_1, rct_2) - impact(rct_2, rct_1)|$.

Example 23 *Suppose we have two couples of unranked resource constraints* $\{rct_A, rct_B\}$ *and* $\{rct_C, rct_D\}$ *such that:*
$impact(rct_A, rct_B) = 10$, $impact(rct_B, rct_A) = 10$, *and*
$impact(rct_C, rct_D) = 20$, $impact(rct_D, rct_C) = 100$.
It means that we would first select the couple $\{rct_C, rct_D\}$ *and choose the decision* $rct_C \rightarrow rct_D$. *After this decision has been propagated, it may be that it will be easier to choose between* $rct_A \rightarrow rct_B$ *and* $rct_B \rightarrow rct_A$.

We see that until now, our criterion to select a potential conflict is very local. A way to improve this is to prefer the potential conflicts that belong to a part of the schedule where there is still a lot of unresolved potential conflicts. This strategy has a double advantage:

1. We can hope that the resolution of such conflicts will have globally less impact as they belong to a rather relaxed part of the schedule and,
2. It ensures a more homogeneous repartition of the solved potential conflicts so that the local criterion to measure the impact of a decision is more accurate.

The criterion we use to estimate whether there are still a lot of potential conflicts around a couple $\{rct1, rct2\}$ is the minimum between the number of resource constraints that are still unranked with respect to rct_1 ($nbUnranked(rct_1)$) and the number of resource constraints that are still unranked with respect to rct_2 ($nbUnranked(rct_2)$). Thus, we use the following criterion as a measure of the opportunity of the potential conflict $\{rct1, rct2\}$:

$$min(nbUnranked(rct_1), nbUnranked(rct_2)) * |impact(rct_1, rct_2) - impact(rct_2, rct_1)|$$

We have compared the quality of the first solution (makespan) found by $LEASTCOM$ on jobshop problems with some state of the art greedy algorithms and dispatching rules. The following table summarize these results[19]. The first column represents the instance of the jobshop problem. The column BLB represents the best known lower bound on the optimal makespan while the column BUB represents the best known upper bound on the optimal makespan. For closed problem, we only give BLB as $BLB = BUB$. Column A shows the makespan of the first solution found by the goal $LEASTCOM$ described in this section. The relative distance of this makespan to the BLB is given on column $\%A$. We see that in average, the makespan of the first solution is 7.35% higher than the BUB. This result is to be compared with the performances of the goal $AMCC$ described in [9] (in columns B and $\%B$), the goal $GREEDY$ described in [1] (in columns C and $\%C$) and the predefined ILOG Scheduler goal $IlcSetTimes$ (in columns D and $\%D$). We can see that $LEASTCOM$ outperforms all of them with an improvement greater than 20% of the average distance to the optimal.

[19] These results were obtained with an edge-finder at level 2 and with the constraint described in section 3.4.1

PB	BLB	BUB	A	% A	B	%B	C	% C	D	% D
abz5	1234	1234	1304	5.67%	1346	9.08%	1330	7.78%	1334	8.10%
abz6	943	943	1062	12.62%	985	4.45%	1052	11.56%	989	4.88%
abz7	656	656	722	10.06%	753	14.79%	781	19.05%	764	16.46%
abz8	645	669	728	12.87%	833	29.15%	763	18.29%	791	22.64%
abz9	661	679	785	18.76%	801	21.18%	810	22.54%	860	30.11%
ft06	55		59	7.27%	55	0.00%	55	0.00%	60	9.09%
ft10	930		1008	8.39%	1007	8.28%	1013	8.92%	1090	17.20%
orb1	1059		1159	9.44%	1233	16.43%	1168	10.29%	1265	19.45%
orb2	888		1020	14.86%	948	6.76%	952	7.21%	1033	16.33%
orb3	1005		1178	17.21%	1162	15.62%	1199	19.30%	1283	27.66%
orb4	1005		1104	9.85%	1108	10.25%	1158	15.22%	1112	10.65%
orb5	887		988	11.39%	924	4.17%	957	7.89%	1030	16.12%
orb6	1010		1089	7.82%	1135	12.38%	1226	21.39%	1277	26.44%
orb7	397		437	10.08%	440	10.83%	463	16.62%	473	19.14%
orb8	899		1060	17.91%	986	9.68%	1020	13.46%	1049	16.69%
orb9	934		1062	13.70%	1015	8.67%	995	6.53%	1144	22.48%
orb10	944		1035	9.64%	1030	9.11%	1117	18.33%	1149	21.72%
la01	666		667	0.15%	666	0.00%	698	4.80%	671	0.75%
la02	655		703	7.33%	730	11.45%	699	6.72%	835	27.48%
la03	597		630	5.53%	735	23.12%	683	14.41%	696	16.58%
la04	590		617	4.58%	637	7.97%	687	16.44%	696	17.97%
la05	593		593	0.00%	593	0.00%	595	0.34%	593	0.00%
la06	926		926	0.00%	926	0.00%	926	0.00%	926	0.00%
la07	890		890	0.00%	984	10.56%	890	0.00%	960	7.87%
la08	863		873	1.16%	904	4.75%	868	0.58%	893	3.48%
la09	951		951	0.00%	999	5.05%	951	0.00%	951	0.00%
la10	958		958	0.00%	1026	7.10%	962	0.42%	958	0.00%
la11	1222		1222	0.00%	1286	5.24%	1257	2.86%	1222	0.00%
la12	1039		1039	0.00%	1075	3.46%	1039	0.00%	1050	1.06%
la13	1150		1150	0.00%	1214	5.57%	1150	0.00%	1189	3.39%
la14	1292		1292	0.00%	1292	0.00%	1292	0.00%	1292	0.00%
la15	1207		1209	0.17%	1406	16.49%	1239	2.65%	1390	15.16%
la16	945		1005	6.35%	979	3.60%	1034	9.42%	1073	13.54%
la17	784		817	4.21%	800	2.04%	836	6.63%	845	7.78%
la18	848		942	11.08%	916	8.02%	913	7.67%	983	15.92%
la19	842		917	8.91%	846	0.48%	966	14.73%	1000	18.76%
la20	902		912	1.11%	930	3.10%	970	7.54%	984	9.09%
la21	1046		1133	8.32%	1220	16.63%	1211	15.77%	1194	14.15%
la22	927		1037	11.87%	1072	15.64%	1189	28.26%	1084	16.94%
la23	1032		1054	2.13%	1122	8.72%	1124	8.91%	1155	11.92%
la24	935		1049	12.19%	999	6.84%	1017	8.77%	1085	16.04%
la25	977		1066	9.11%	1071	9.62%	1168	19.55%	1112	13.82%
la26	1218		1371	12.56%	1324	8.70%	1218	0.00%	1386	13.79%
la27	1235		1346	8.99%	1420	14.98%	1466	18.70%	1413	14.41%
la28	1216		1292	6.25%	1365	12.25%	1354	11.35%	1468	20.72%
la29	1142	1153	1359	19.00%	1438	25.92%	1428	25.04%	1352	18.39%
la30	1355		1447	6.79%	1497	10.48%	1512	11.59%	1577	16.38%
AVERAGE				7.35%		9.33%		9.95%		13.20%
MAX				19.00%		29.15%		28.26%		30.11%
VARIANCE				0.32%		0.47%		0.61%		0.69%
AVERAGE DEV				4.69%		5.14%		6.51%		6.79%
STANDARD DEV				5.70%		6.82%		7.80%		8.32%

Chapter 4

Conclusion and Future Works

This report firstly introduces a new concept, the one of Modal Precedence Graph that extends classical Precedence Graphs in two directions: (1) the representation of *next* relations and (2) the representation of *possibly contributing* vertices. These extensions are very useful for practical Scheduling problems. The second part of this report deals with the integration of MPG into ILOG Scheduler 4.4. We describe the general integration architecture as well as some algorithms that link MPG with the rest of ILOG Scheduler. As we have seen in this report, the interest of using MPG in Scheduler are the following:

- In general, MPG allow to improve the performances of the Disjunctive Constraint. This improvement is very sensible in the lowest part of the search tree when some resources are almost ranked (see figure 3.3). Notice anyway that this improvement is not systematical.
- MPG allow to improve the performances of Sequence Constraint for about 40% (see figure 3.4). That's the main reason why, in Scheduler 4.4, a MPG Constraint is automatically created when a Sequence Constraint is defined.
- MPG allows to improve the propagation of Sequence Constraint as seen in section 3.3.3. This extra-propagation permits to sensibly reduce the domains of the previous and next variables when the density of temporal constraints is high.
- MPG can be seen as a light version of the Sequence Constraint when one just need to express and propagate some next relations between resource constraints. In that case, the MPG constraint alone is much quicker than the Sequence Constraint (see figure 3.2).
- For problems with precedence constraints and transition times, the MPG allows to propagate the transition times along the precedence constraints. This extra-propagation may save a lot of search efforts as seen on the $trolley2$ example in section 3.2.5.
- As shown in section 3.4, MPG allows to write new powerful propagation algorithms that reasons on the relative position of activities on a resource. Most of the categories of resources may benefit from such constraints (unary, discrete, energy resources, reservoirs). Another example will be found in [2] on TSP with Time Windows.
- MPG allow the implementation of new easy-to-write and efficient search goals. The goal $LEASTCOM$ of section 3.5.1 that outperforms the state of the art greedy algorithms and dispatching rules to find a good first solution for jobshop problems is particularly illustrative.

- Some problem decomposition strategies may also benefit from MPG . For example, MPG allows to easily detects critical activities as described in [6], Chapter ? .
- We can also notice that MPG are an interesting structure to store some partial or complete solutions for local search or repair (see for example the use of MPG in the class `IlcSchedulerSolution` [5].)

Future work on MPG will consists in:

- Studying new constraints and search goals relying on MPG .
- Introducing on the MPG the notion of minimal and maximal position of a vertex. This notion has two interests. Firstly, it has a practical semantics and it could give the user the possibility to easily express constraints like *the difference between the position of activity A and the position of activity B on the resource must be at least k.* Secondly, this notion can be used to improve the performances and the memory consumption of MPG as it easily allows a decomposition of the MPG into subsets of vertices that are completely ranked with respect to each other.
- Studying other possible representations of the MPG in order to save both memory and time. Possible interesting candidates are the interval and the chain approaches as described in [8].

Appendix A

Proofs

A.1 Theorem 2

Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a consistent MPG . There exists a unique MPG $G^+ = (V, \mathcal{C}^+, \mathcal{S}^+, \mathcal{N}^+)$ such that G^+ is equivalent to G and G^+ dominates all the graph in the equivalence class of G. G^+ is called the **closure** *of G.*

Proof:

Let $G = (V, \mathcal{C}, \mathcal{S}, \mathcal{N})$ be a consistent MPG . As G is consistent, $\Pi(G) \neq \emptyset$.
Let $G_0 = (V, \mathcal{C}_0, \mathcal{S}_0, \mathcal{N}_0)$ be the MPG such that:

- $\mathcal{C}_0$ is defined as follows, $\forall v \in V$:
  - $\mathcal{C}_0(v) = \{0\}$ if and only if $\{\pi \in \Pi(G) / v \in \pi\} = \emptyset$
  - $\mathcal{C}_0(v) = \{1\}$ if and only if $\forall \pi \in \Pi(G), v \in \pi$
  - $\mathcal{C}_0(v) = \{0, 1\}$ otherwise
- $\mathcal{S}_0$ is defined as follows, $\forall (v, w) \in V \times V$, $(v, w) \in \mathcal{S}_0 \Leftrightarrow w \in Succ_G(v)$.
- $\mathcal{N}_0$ is defined as follows, $\forall (v, w) \in V \times V$, $(v, w) \in \mathcal{S}_0 \Leftrightarrow w \in Next_G(v)$.

The definition of G_0 from G is not ambiguous.

Example 24 *We have seen that the two graphs represented on figure 2.7 were equivalent as they represent the same set of topological sorts $\{(v_0, v_1, v_2, v_3), (v_0, v_1, v_3, v_2)\}$. If we apply the construction process described above to these graphs, we obtain the graph G_0 represented on figure A.1*

Now we will show that:

1. $\Pi(G) \subset \Pi(G_0)$
2. G_0 dominates G
3. G_0 dominates all the MPG that are equivalent to G
4. if G_1 is a MPG equivalent to G_0 such that G_1 dominates G_0, then we have $G_1 = G_0$.

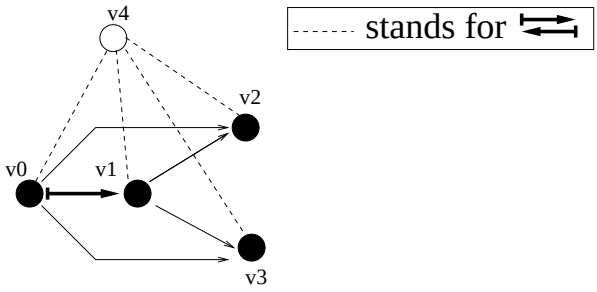

Figure A.1: Prooving Existence and Unicity of the Closure

Notice that the fact that G_0 and G are equivalent is a consequence of points 1 and 2.

Point 1. Let π be a topological sort of G. We want to show that π is also a topological sort of G_0. This is true because:

1. if $v \in \pi$, it follows from the definition of $\mathcal{C}_0$ that $\mathcal{C}_0(v) \neq \{0\}$ and thus we have $1 \in \mathcal{C}_0$.
2. if $v \notin \pi$, it follows from the definition of $\mathcal{C}_0$ that $\mathcal{C}_0(v) \neq \{1\}$ and thus we have $0 \in \mathcal{C}_0$.
3. let $(v, w) \in \mathcal{S}_0$ such that $v \in \pi$ and $w \in \pi$. By definition of $\mathcal{S}_0$, we have $w \in Succ_G(v)$. As we have $w \in Succ_G(v)$, $v \in \pi$ and $w \in \pi$, it means that $pos(w, \pi) > pos(v, \pi)$.
4. let $(v, w) \in \mathcal{N}_0$ such that $v \in \pi$ and $w \in \pi$. By definition of $\mathcal{N}_0$, we have $w \in Next_G(v)$. As we have $w \in Next_G(v)$, $v \in \pi$ and $w \in \pi$, it means that $pos(w, \pi) = pos(v, \pi) + 1$.

Point 2. Let's show that G_0 dominates G.

1. Let $(v, w) \in \mathcal{S}$, we must show that $(v, w) \in \mathcal{S}_0$. If $(v, w) \in \mathcal{S}$, property 1 states that $w \in Succ_G(v)$. Thus, as w is a successor of v on G, by construction of G_0, $(v, w) \in \mathcal{S}_0$.
2. Let $(v, w) \in \mathcal{N}$, we must show that $(v, w) \in \mathcal{N}_0$. If $(v, w) \in \mathcal{N}$, property 1 states that $w \in Next_G(v)$. Thus, as w is next to v on G, by construction of G_0, $(v, w) \in \mathcal{N}_0$.
3. Let $v \in V$, we must show that $\mathcal{C}_0(v) \subset \mathcal{C}(v)$. Suppose that $\{0\} \subset \mathcal{C}_0(v)$ and $\mathcal{C}(v) = \{1\}$. This is clearly impossible because in that case, $\forall \pi \in \Pi(G), v \in \pi$ and thus, $\mathcal{C}_0(v) = \{1\}$. Suppose that $\{1\} \subset \mathcal{C}_0(v)$ and $\mathcal{C}(v) = \{0\}$. This is clearly impossible because in that case, $\forall \pi \in \Pi(G), v \notin \pi$ and thus, $\mathcal{C}_0(v) = \{0\}$. Thus, $\mathcal{C}_0(v) \subset \mathcal{C}(v)$.

Point 3. Let's show that G_0 dominates all the MPG that are equivalent to G.

Let G' be a MPG equivalent to G. We have $\Pi(G') = \Pi(G)$. As you can notice, the process we used to construct G_0 from G only depends on the set of topological sorts of G. Thus, if we apply the same process to create the MPG G'_0 from G', we will get the same resulting graph $G'_0 = G_0$. It follows from point 2 that G'_0 dominates G', thus, G_0 dominates G'.

Point 4. We will now show that if G_1 is a MPG equivalent to G_0 such that G_1 dominates G_0, then we have $G_1 = G_0$.

Suppose that we have $G_1 = (V, \mathcal{C}_1, \mathcal{S}_1, \mathcal{N}_1)$, $G_1 \equiv G_0$ and that G_1 dominates G_0 and $G_1 \neq G_0$. It means that we are in at least one of the following cases:

1. either $\exists (v, w) \in \mathcal{S}_1 / (v, w) \notin \mathcal{S}_0$

2. or $\exists (v,w) \in \mathcal{N}_1 / (v,w) \notin \mathcal{N}_0$
3. or $\exists v \in V / \mathcal{C}_1(v) = \{0\}\ and\ \mathcal{C}_0(v) = \{0,1\}$
4. or $\exists v \in V / \mathcal{C}_1(v) = \{1\}\ and\ \mathcal{C}_0(v) = \{0,1\}$

Case 1 is not possible because in that case, we would have $w \in Succ_{G_1}(v)$ and thus $w \in Succ_{G_0}(v)$ (as G_0 and G_1 are equivalent) so that, by construction, $(v,w) \in \mathcal{S}_0$.

Case 2 is not possible because in that case, we would have $w \in Next_{G_1}(v)$ and thus $w \in Next_{G_0}(v)$ (as G_0 and G_1 are equivalent) so that, by construction, $(v,w) \in \mathcal{N}_0$.

Case 3 is not possible because in that case we would have $\{\pi \in \Pi(G_1) / v \in \pi\} = \emptyset\}$ and thus, $\mathcal{C}_0(v) = \{0\}$.

Case 4 is not possible because in that case we would have $\forall \pi \in \Pi(G_1), v \in \pi$ and thus, $\mathcal{C}_0(v) = \{1\}$.

End of proof.

Points 1, 2 and 3 show that G_0 is a MPG that is equivalent to G and dominates all the MPG in the equivalence class of G.

Point 4 shows that it is the unique MPG with such a property. Indeed, if there was another MPG G_1 that is equivalent to G and dominates all the MPG in the equivalence class of G, then, G_1 would be equivalent to G_0 and would dominate it so that, according to point 4, $G_1 = G_0$.

□□□

Bibliography

[1] Y. Caseau and F. Laburthe. Disjunctive scheduling with task intervals. Technical report, LIENS Technical Report 95-25, École Normale Supérieure Paris, France, July 1995.

[2] F. Focacci, A. Lodi, and M. Milano. Solving TSP with Time Windows with Constraints. In *CP'99 (submitted to)*, 1999.

[3] ILOG S.A. *ILOG Scheduler 4.3, Reference Manual*.

[4] ILOG S.A. *ILOG Scheduler 4.3, User's Manual*.

[5] ILOG S.A. *ILOG Scheduler 4.4, Reference Manual*.

[6] ILOG S.A. *ILOG Scheduler 4.4, User's Manual*.

[7] P. Laborie and M. Ghallab. Planning with Sharable Resource Constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1643–1649, Montreal (Canada), August 1995.

[8] Esko Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Helsinki University of Technology, 1995.

[9] D. Pacciarelli and A. Mascis. Job-Shop Scheduling of Perishable Items. In *INFORMS*, Cincinnati (USA), 1999.