Imperial College of Science, Technology and Medicine

(University of London)

Department of Computing

Constraint-Logic Based Approach To Scheduling

by

Nicolaos Angelopoulos

Submitted in partial fulfilment of the requirements for the MSc degree in Engineering of the University of London and for the Diploma of Imperial College of Science, Technology and Medicine.

September 1993

Acknowledgements

I would like to thank Professor Kowalski for his guidance and encouragement throughout the duration of this work. Particularly for communicating his scientific ideas and for spending time over my full-of-typographical-mistakes drafts.

Also I would like to thank Francesca Tony for her comments on earlier fragments of this work; from which a number of improvements have been accomplished.

Finally the reader is reminded that all mistakes, misconceptions, inconsistencies and shortcomings of this report are, exclusively, due to the author's own intellectual limitations.

It's *cool* to be *cooking with gas* - even if only for a while.

Contents

List of Figures

List of Tables

Abstract

This work is an attempt to bring forward and amalgamate a number of different technologies. Our motivation originates in extending the classical Logic Programming paradigm for effective reasoning over disjunctive information. In doing so we take aboard notions from the area of constraint satisfaction. We generalise these notions by allowing user-defined constraint rules. The level of programming required from the user is further abstracted by allowing experimentation with different control strategies. The resulting framework although in certain respects enhances Logic Programming with Constraints, it still is quite distinct to other approaches which are termed as Constraint Logic Programming. Here scheduling is used as a means for illustrating the framework's potentialities. Our choice is based on the abundance with which disjunctive information occur in the particular area.

Schedule : akin to Greek $\sigma\chi\acute{\iota}\zeta\epsilon\iota\nu$, to split.

Chapter 1

Introduction

Our embarking point for this intellectual excursion is Logic Programming; with the objective of enhancing its up-to-date realisations for effective reasoning over disjunctive information. The method is by incorporating constraints as a means of replacing, in certain instances, backtracking (and its maladies, as discussed in [Mac77]) by forward reduction of the search space. The resulting marriage is a particularly convenient one (apart from being a love affair in the author's scientific understanding) since constraints have a dual status, on one hand they are just a special subclass of relations and also, they can provide an abstract means for allowing user-defined control (which in itself is a route for producing new algorithms, when in logical settings, as argued in [Kow79]).

The contrived artifact is then applied to scheduling examples. Scheduling problems have been a fertile area of research for a number of years. The main fascinating point being that although one may express these problems quite succinctly, the search for a solution, more often than not, is characterised as an NP-hard problem. Thus a number of compromising locations have been proposed on where to draw the line between efficiency of computation and optimality of solution (eg [SWV92] [ABZ88]). A second reason for the copious labours in the area is its relevance to practical applications; particularly so in the contemporary society's drive for automation and reduced production costs.

Outline of intentions

Here we attempt to give an intuitive feel for the novel points of the proposed framework and how they may be used to explore new possibilities in the area of scheduling.

The first distinguishing point for our approach is that one reasons at the level of completion (ie the completed database). Thus for each predicate-definition the user is expected to present its completed definition. This requires the extension of clausal form, which is used in Logic Programming.

The second discerning issue is the utilisation of user-defined Constraint Rules. One view is that these rules are related to consistency in a manner analogous to the way Logic Programming relates to deduction. In particular they provide an escape route for eluding the trying limits of finite domains; which have bound most of the previous research on constraint reasoning.

By combining the above two we overcome some of the barriers which were previously confining them. More specifically the *blind* chronological backtracking has been avoided. The main mechanics for achieving this is by delaying the commitment of choosing between consistent disjuncts. Furthermore constraint rules are used for determining the consistency of the various disjuncts while at the same time information is accumulated about their consistency with other uncommitted (pending) disjunctions. Thus in effect the framework is capable of efficiently interleaving depth-first search with breadth-first techniques.

A further possibility, arising from using the proposed framework; and of particular interest to scheduling, is the new search spaces it unveils. A new set of heuristics are made accessible for replacing backtracking and problem specific heuristics. The new heuristics can be indexed to either of two groups. On one hand we have to control the aforementioned interleaving specifics (trading one search technique for the other) and on the other hand we can use the results of propagation[1] for making intelligent guesses.

Finally the notion of control attracts a substantial amount of the dissertation's attention. Our thesis being that definition of control is not necessarily

[1]propagation : use of constraint rules for extracting information

a burden to the process of problem solving. On the contrary once we separate control issues from the logical ones and provide some abstract means for their definition we arrive to particularly useful computation formalisms. In this framework Constraint Rules are such abstract means, which are complemented by definitions of the two classes of heuristics, as mentioned in the previous paragraph.

Related work

In this section we present the areas and specific references to which our work is closely related. Of course not all areas are equally-distant relatives.

The highlighting overlaps with Logic Programming are issues of knowledge representation and the program completion. The main antithesis is our explicit treatment of disjunctions and the direct (rather than via metaprogramming) access to control provided to the user. The Logic Programming perspective of these issues are covered in [Hog90]; to which we base our comparative comments.

The uses and significance of disjunctive constraints have been perceived in a number of otherwise unrelated works. These include [SH92], [VHD91] and [Wal93]. Also surprisingly enough the concepts and intuitions we have developed for disjunctive constraints and their handling are close to the work done on the AKL parallel Constraint Logic Programming (CLP) language (see [Har90]), although their motivation is quite different and their approach to consistency is a more conventional one.

Our principal ideas on constraint reasoning have been greatly influenced by [Mac77]. The presented framework can be viewed as an attempt to generalise the results of this paper; generalise with respect to the applicable domains.

Notions which are very close to what in this work are called Constraint Rules appear in the recent work of [FH93]. Similar concepts can be found in constraint-related graph rewriting approaches.

Although there exists a rich body of literature on Constraint Logic Programming systems and their application to real-life problem solving, these systems are still far from being used by the uninitiated programmer/user.

The main criticism being that the wanton programmer has to discipline herself in very strict ways if she is to use the systems advantageously. Furthermore the basic, built-in, constraints which are the only ones provided by the current CLP systems are not guaranteed to be appropriate for anything else but a limited number of applications. Such systems include [Col90] [DHS+88] [JM87] and [JL87] (the latter of which is a schema for CLP but for which the above mentioned restrictions still hold).

Chapter 2

Syntax

Here we present the syntactic objects of the framework. We appeal to simplicity and intuition rather than to strict formal definitions. Following the initial formulation of syntactic notions, is a simple example through which we illustrate the representation of different aspects arising in scheduling settings.

2.1 Syntactic Objects

Our approach gives two main handles to the user; which are used for knowledge representation and for guiding derivation. These are *iff-definitions* and *constraint rules*. Each having a precise syntax and a unique role in the computation process. In what follows we expose their syntax. Also we clarify the terminology adopted in this work.

2.1.1 basics

We adopt classical first order logic syntactic conventions. So by $\wedge$ and $\vee$ we, respectively, refer to the logical operators AND and OR. While atomic formulae are formed from a predicate symbol (of a given arity) and a number of associated terms (here we will often refer to atomic formulae as predicates). Each term constructed recursively, in the usual manner, from the sets of constants, function symbols and variables. A formula is analogous to a classical logic formula but for the operators allowed, which are restricted to $\wedge$ and $\vee$. Quantifiers are treated implicitly, a la Logic Programming, thus they do not appear at the syntactic level. By *axioms* we refer to a set of predefined predicates.

2.1.2 iff-definitions

Knowledge representation in the framework is via *iff-definitions*; which are user defined and have the form,

$$S\ iff\ C_1 \vee C_2 \vee \ldots \vee C_n.$$

Where each of the $C_i's$ is a conjunction of predicates, and S is a single predicate. We refer to S as a *user-predicate*; with the particular iff-definition called the definition of S. Any set of iff-definitions (representing a *program*) should only have at most one definition for any predicate appearing in it. Each such definition should be complete in its description of the defining user-predicate.

2.1.3 constraint rules

Constraint rules are also user defined but they are, contrary to *iff-definitions*, more independent from any specific problem we may wish to solve. (Although specific rules may come into play for solving problems belonging to certain subclasses; as to benefit from their special characteristics.) Instead for each area (scheduling being one of them) there will be a certain number of basic constraint rules which will be responsible for the main body of constraint reasoning. There are two classes of such rules.

Addition rules have the general form $\alpha \rightsquigarrow \beta$ with the intuition that whenever we have α we may add to our constraints β. Where α has the form $\alpha_1 \wedge \alpha_2$. With α_2 being an arbitrary formula of *axioms* and α_1 an arbitrary formula of *constraints* [1]. Predicates appearing in β can be either *constraints* or *user-predicates*.

Similarly *deletion rules* are of the form $\gamma \wedge \delta \hookrightarrow \delta$ and are used for deleting δ whenever we have $\gamma \wedge \delta$. The formula describing γ can include both axioms and constraints, whereas δ is a conjunctive formula of *constraints*.

[1] implicitly defined as pointed in what follows

The above constitute an implicit definition for the set of *constraints*. In more explicit terms we may say that all predicates appearing in α, β, γ and δ which are not *axioms* (or *user-predicates* in the case of β) are, in fact, *constraints*. Another significant point, which follows from the above, is that there is not clear cut distinction between *user-predicates* and *constraints* (as shown in Figure 2.1). Further to this we allow for predicates to be characterised as either of the two, depending on the way the particular formula, in which they appear, is to be used. This is in contrast to most of the current approaches to constraint logic programming. In our presentation there have be only one instance for which the intersection of the two sets was other than the empty set. But in other areas, such as planning, their may exist a substantial overlap. Such an approach has been pursued in [Her93].

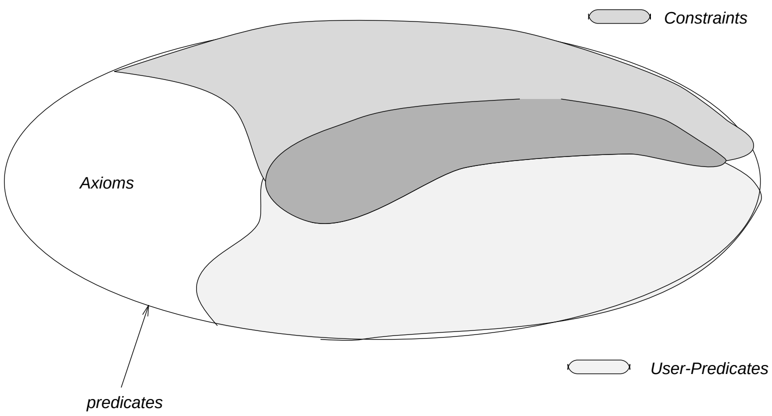

Figure 2.1: Predicate's Classification

2.2 An Introductory Example

Here we use a critical path problem for demonstrating the representation of several aspects of scheduling problems. We start with the simple case where only precedence constraints are present and then comment on the more elaborate cases.

The standard algorithm can be found in [Kri84]. Where a prolog-based solution is presented. Also [VH89] discuss both the simple case and some more advanced examples.

2.2.1 specifying the problem

In the simplest case we have a number of tasks. Each task having a known duration and a number of (precedence) constraints (ie some tasks can only be performed after the completion of, certain, other tasks). Our goal is to find a schedule which will minimise the finishing time (of the latest task). This is the well known critical path problem. Where one is concerned with finding the tasks which are critical to the overall execution of tasks. (Critical as to that any extra delay on the execution of the particular task will be responsible for delaying the overall finishing time.)

name	duration	prec. tasks
A	4	-
B	13	A
C	4	-
D	15	C
E	12	-
F	4	A
G	2	E,F
H	2	B,D
I	2	G,H

Table 2.1: A Critical Path Problem

A specific problem in this class is presented in Table 2.1, taken from [Kri84] (a network representation appears in Figure 2.2). The standard Operations Research (OR) solution uses a labeling algorithm and can be viewed as a three stage process :

1. forward labeling, of earliest start times,
2. forcing earliest start time, of last[2] task, to also be its latest start time,
3. backward labeling of latest start times.

Which correspond to the three respective parts of Figure 2.3.

[2]last here denotes the task with the maximum, earliest start date

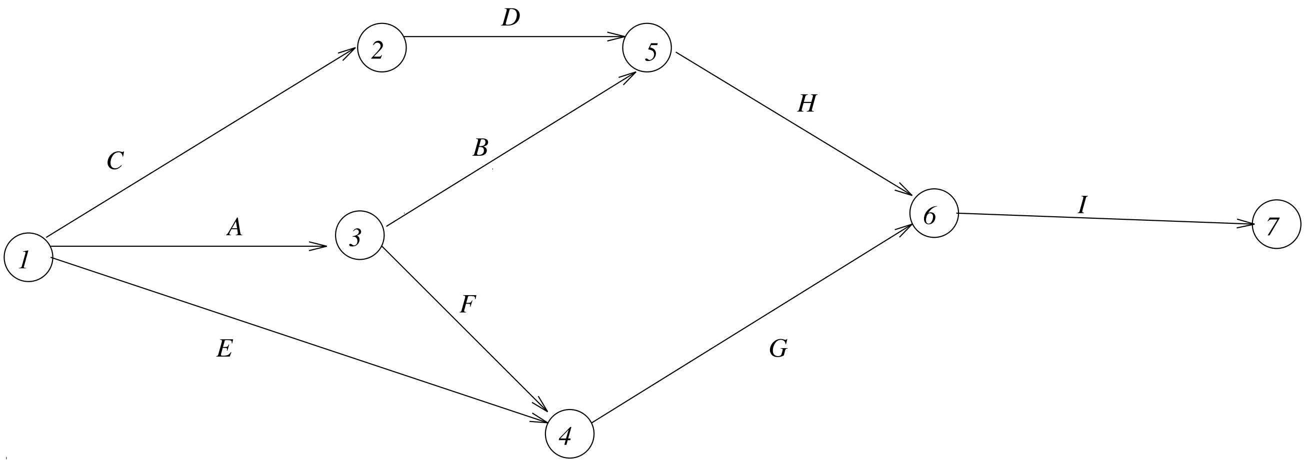

Figure 2.2: Network Representation

2.2.2 representation within the Framework

From the preceding exposition one can see that the OR algorithm uses the network representation as to reason about start times relative to *stages* rather than to tasks. (Naturally enough, these stages are dependent to the task structure.) Where a stage is a single node in the graph representation (Figure 2.2).

In the proposed framework one reasons at the level of tasks. Precedence constrains are interpreted as constraints on task starting times. So if we let t_a, t_b denote the start times of tasks A and B respectively, we can then capture the constraint A < B by $t_a + 4 \leq t_b$. Thus by using simple propagation rules we can map the forward labeling to a number of propagation steps (eg since $C < D < H$ which is $0 \leq t_c, t_c + 4 \leq t_d$, $t_d + 15 \leq t_h \overset{*}{\rightsquigarrow} 19 \leq t_h$ which corresponds to the label given at node 4; and even if this was not the case it would still have, in all cases, be considered as a candidate for the label).

Similarly the backward labeling corresponds to the propagation steps initiated by the upper bound on task's I starting time. Thus from B < H < I (ie $t_b + 13 \leq t_h, t_h + 2 \leq t_i \overset{*}{\rightarrow} t_h \leq 19$, $t_b \leq 6$) one can propagate values from I to H and from there to B.

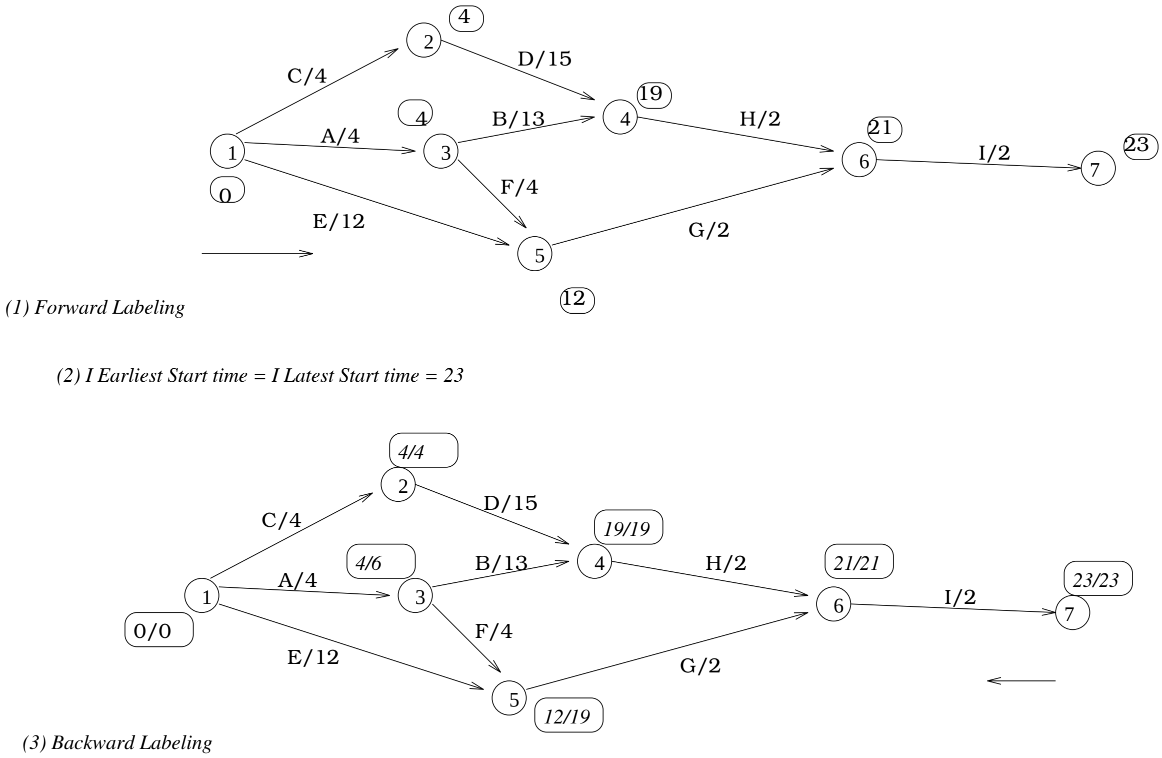

Figure 2.3: The Labeling Process

2.2.3 Framework's solution (basics)

iff-definitions

$$
\begin{array}{ll}
\text{start\_times iff} & 0 \leq t_a \wedge 0 \leq t_c \wedge 0 \leq t_e \\
\text{precedence iff} & t_a + 4 \leq t_b \wedge t_c + 4 \leq t_d \wedge t_a + 4 \leq t_f \\
& \wedge t_f + 4 \leq t_g \wedge t_e + 12 \leq t_g \wedge t_b + 13 \leq t_h \\
& \wedge t_d + 15 \leq t_h \wedge t_g + 2 \leq t_i \wedge t_h + 2 \leq t_i
\end{array}
$$

constraint rules

$$W \leq t_X \wedge t_X + L \leq t_Y \wedge VisL + W \rightsquigarrow V \leq t_Y$$
$$t_Y \leq W \wedge t_X + L \leq t_Y \wedge VisW - L \rightsquigarrow t_X \leq V$$
$$V \leq t_X \wedge V < 0 \rightsquigarrow false$$

$t_X \leq V \wedge t_X \leq W \wedge V < W \hookrightarrow t_X \leq V$
$V \leq t_X \wedge W \leq t_X \wedge V < W \hookrightarrow W \leq t_X$

query

$$start\_times \wedge precedence \wedge t_i + 2 \leq 23.$$

the solution

The complete solution, as given by propagation, follows in Table 2.2. Obviously not all of the given intervals can be satisfied, in any arbitrary way. But we are guaranteed a consistent assignment when all tasks start at their earliest possible.

From the above, one can see the similarity in complexity and operation of the two approaches (the difference being that the first needs programming at the labeling-nodes level while the other requires the declaration of precedence constraints and local propagation rules). What is not so direct in our approach is how we match the second step of the OR algorithm. This will become clearer once the discussion on controlling derivation has taken place (Section 4.1). Here we have simplify matters by making the query more specific. One may perceive it as the questioning of whether a solution exist that achieves a time less than or equal to 23 time units.

lb on st	$\leq$ task s.t. $\leq$	ub on st
0	A	2
4	B	6
0	C	0
4	D	4
0	E	7
4	F	15
12	G	19
19	H	19
21	I	21

Table 2.2: Permissible Intervals

2.3 Advanced Representation Issues

Further to this simple example real life problems also incorporate different species of constraints. Here we turn our attention on handling two such categories of constraints.

2.3.1 technological constraints

Technological constraints include a number of quantitative relations between different tasks. Such constraints are normally dictated by the technology governing the domain of discourse. For example in the presented problem tasks could be the ones required for building a house. With A be the task of excavating and B of setting the foundations. Due to external factors, we may have to constraint the two tasks to be at the most 5 time units apart (with A coming before B). Similarly tasks E and G may have to be at least 2 units apart.

Although the qualitative part of these constraint is already accounted for in the OR approach, still there exist no straightforward way for incorporating the quantitative restrictions. On the contrary CLP can readily accommodate them and in many cases these, extra constraints, make problems *easier* to solve. Furthermore in our approach there is no conceptual price we have to pay. The two pieces of information are readily translated to $t_A + 6 \leq t_B$ and $t_E + 14 \leq t_G$. While no extra effort needed for incorporating them into our reasoning process.

2.3.2 disjunctive constraints

Disjunctive constraints are a major point for comparison and discussion about the proposed framework. Particularly so since disjunctive constraints have been used in various Constraints-in-Logic Programming approaches (such as [Wal93] [SH92] and [VHD91]). The common point of reference for these approaches is that disjunction is handled implicitly. In [VH89] this is done by contriving a data representation, while the other approaches give a special control interpretation to some syntactic objects (either a predicate or some new logical operator).

Here we try to capitalise on the usefulness of disjunctive constraints by representing them explicitly. Our view is that since constraints are relations (evidently, of a rather specific nature) we treat them indistinguishably at the representation level. Their uniqueness is exploited through the use of constraint rules and different control strategies.

In the particular example we may have constraints regarding the use of resources. For instance tasks C and F may require the exclusive use of a singly available resource (eg a crane). So it will either be C that is executed first or task F will precede task C. In the proposed framework one may express this information using the disjunction, $t_c + 4 \leq t_f \vee t_f + 4 \leq t_c$.

Chapter 3

Semantics

We now turn our attention towards the semantics attached to the introduced syntax; doing so by virtue of the operational characterisation of the computation sequence. We also present the requirements (on constraint rules) which can guarantee mathematically clean association of the problem solution to the problem statement. Closing this chapter is a brief presentation of a more general framework, from which the presented framework can be derived as a particular instance.

3.1 The Computation Sequence

We view a computation as a sequence of derivation steps. Each step derived from its predecessor according to one of a number of possible derivation rules.

In what follows let :

$A,\ C$	: conjunctions of constraints,
B	: an arbitrary formula,
D	: a conjunction of formulae,
E	: a disjunction of formulae,
S	: a single user-predicate,
X	: an arbitrary formula of axioms, (eg $2 < 3 \wedge \ldots$)
DT	: domain theory, comprising
	iff_clauses, constraint rules and axioms,
G_i	: the query at step i of the computation,

$\vdash_{\mathcal{F}}$: framework derivability; to be defined.

Also, $\vdash_{\mathcal{F}}$ and $\dashv_{\mathcal{F}}$ are distinct and equivalence preserving (provisionally so, as discussed in the following sections).

$$
\begin{array}{lll}
\multicolumn{3}{l}{G_i \vdash_{\mathcal{F}} G_{i+1} \text{ iff}} \\
& \multicolumn{2}{l}{G_i = [(A \vee B) \wedge C \wedge D] \vee E \ \&} \\
& & G_{i+1} = (A \wedge C \wedge D) \vee (B \wedge C \wedge D) \vee E \\
& \text{or} & G_{i+1} = [(A \vee B) \wedge B' \wedge D] \vee E \ \&\ C \dashv_{\mathcal{F}} B' \\
& \text{or} & G_{i+1} = [(B' \vee B) \wedge C \wedge D] \vee E \ \&\ A \wedge C \dashv_{\mathcal{F}} B' \\
& \text{or} & G_{i+1} = [(E' \vee B) \wedge C \wedge D] \vee E \ \& \\
& & \qquad\qquad A = S \ \&\ (S \ iff\ E' \in DT \\
& \text{or} & G_{i+1} = (B \wedge C \wedge D) \vee E \ \& \quad A = \ false \\
& \text{or} & G_{i+1} = E \ \& \quad C = false
\end{array}
$$

Where

$$
\begin{array}{l}
B_1 \wedge \ false \ \dashv_{\mathcal{F}} \ false \\
B_1 \wedge B_2 \wedge X \dashv_{\mathcal{F}} B_1 \wedge B_2 \wedge B_3 \ \text{ if} \\
\qquad\qquad (B_1 \leadsto B_3) \in DT \wedge DT \vdash (X \leftrightarrow \top) \\
B_1 \wedge B_2 \wedge B_3 \wedge X \dashv_{\mathcal{F}} B_1 \wedge B_3 \text{ if} \\
\qquad (B_1 \wedge B_2 \hookrightarrow B_2) \in DT \wedge DT \vdash (X \leftrightarrow \top) \\
B_1 \dashv_{\mathcal{F}} B_3 \qquad \text{if} \qquad B_1 \dashv_{\mathcal{F}} B_2 \ \&\ B_2 \dashv_{\mathcal{F}} B_3
\end{array}
$$

With $G_0 =$ user's query and

$G_n = final\ G_i,\ iff\ \nexists\, G_{n+1}\ st.\ G_n \vdash_{\mathcal{F}} G_{n+1}$

For ease of reference let us name the above disjunctions in the definition of $\vdash_{\mathcal{F}}$ (from top to bottom) by : *split*, *branch propagate*, *leaf propagate*, *unfold*, *efface* and *prune*. (The particular order facilitates a succinct presentation and should not be thought as having any other significance.) The separation of $\dashv_{\mathcal{F}}$ was employed so that properties of the framework will be easier to establish.

In the above we use *false* as a syntactic token for representing inconsistencies in the underlying theory. Thus we anticipate a number of *addition* *rules* (although absence of such rules is perfectly legitimate) which will have the form $\alpha \rightsquigarrow false$, each of which describes an inconsistent formula of constraints. Where underlying theory is the theory of consistency for the particular constraints. Thus by introducing falsity as a result of constraint propagation we can assure that by preserving equivalence we also preserve the consistency of the original formulation (iff-definitions + constraint- rules + query + axioms), provided that the *addition rules* deriving false capture all cases in which inconsistencies (of interest) can be detected. This condition is the price we have to pay for generalising propagation techniques beyond the finite domain settings, to which they are traditionally been restricted.

3.2 A Reading for $\vdash_{\mathcal{F}}$

Given the operational characterisation, we wish to establish the meaning of the framework's derivability through its relation to classical computational logic. In particular we claim that $\vdash_{\mathcal{F}}$ is equivalence preserving when certain properties hold between the underlying theory and the syntactic objects of the framework.

Each of the above possible derivations is either representing a particular choice for efficient computation or describing the effect of syntactic transformation (ie iff definitions are used for unfolding and constraint handling rules for adding and deleting constraints). Since these are two quite distinct categories we treat each of them separately.

3.2.1 syntactic transformations

In the above presentation we defined derivability by a two-layer structure. This improves clarity and captures the differences between reasoning with constraints and reasoning with user-predicates in a orderly way (as oppose to using a single-layer structure). We embark by considering the lower layer.

properties of $\dashv_{\mathcal{F}}$

Our aim here is to show that for the underlying theory (UT) we have, $B \dashv_{\mathcal{F}} B'\ iff\ UT \vdash B \leftrightarrow B'$. In order to do so, we place some requirements on the mathematical nature of the constraint rules.

Addition rules have the general form $\alpha \leadsto \beta$ which reads, that whenever we have α we may add to our constraints β (with the hope that this extra piece of information will lead to either a contradiction or to tighten some other constraints). Here its required that α should imply β, ($\alpha \rightarrow \beta$) in the domain of discourse (with the possibility of using certain axioms of a more fundamental nature; such as arithmetics, inequalities between numbers and membership to the sets of numbers, which in the above presentation are accommodated by the presence of X in the various applications of rules). Thus we require : $UT \vdash \alpha \rightarrow \beta$.

Deletion rules are of the form $\gamma \wedge \delta \hookrightarrow \delta$, used for deleting δ. Similarly to *addition rules* here we require that in the underlying theory we have $UT \vdash \gamma \rightarrow \delta$.

Due to the recursive nature of $\dashv_{\mathcal{F}}$ we prove its equivalence preserving property by means of induction (using three base cases and a recursive one). So we have :

base cases

(a) $B_1 \wedge\ false\ \dashv_{\mathcal{F}}\ false$

trivially $B_1 \wedge\ false\ \leftrightarrow\ false$

(b) $B_1 \wedge B_2 \wedge X \dashv_{\mathcal{F}} B_1 \wedge B_2 \wedge B_3\quad if\ (B_1 \leadsto B_3) \in DT$
$\wedge DT \vdash (X \leftrightarrow \top)$

also we know that $UT \vdash B_1 \rightarrow B_3$, from truth tables we get

$UT \vdash B_1 \wedge B_2 \leftrightarrow B_1 \wedge B_2 \wedge B_3$

(c) $B_1 \wedge B_2 \wedge B_3 \wedge X \dashv_{\mathcal{F}} B_1 \wedge B_3$ if $(B_1 \wedge B_2 \hookrightarrow B_2) \in DT$
$\wedge DT \vdash (X \leftrightarrow \top)$

similarly to the above, here we have, $UT \vdash B_1 \wedge B_3 \rightarrow B_3$

and again by truth tables we get,

$$UT \vdash B_1 \wedge B_2 \wedge B_3 \leftrightarrow B_1 \wedge B_3$$

recursive case

$B_1 \dashv_{\mathcal{F}} B_3$ if $B_1 \dashv_{\mathcal{F}} B_2$ & $B_3 \dashv_{\mathcal{F}} B_4$

where $B_1 \dashv_{\mathcal{F}} B_2$ is syntactic sugar for

any of the base cases, (ie $B_1 \leftrightarrow B_2$ is known)

then by using $B_2 \dashv_{\mathcal{F}} B_3\ iff\ UT \vdash B_2 \leftrightarrow B_3$ as our inductive hypothesis, we directly derive $UT \vdash B_1 \leftrightarrow B_3$

unfolding

A second, admittedly less intriguing, consideration in this section is the *unfolding* operation. *Unfold* requires the existence of $(S\ iff\ E_2) \in DT$. Where the operation replaces S by E_2. The way equivalence is preserved here is by requiring $UT \vdash S \leftrightarrow E_2$. This is, indeed, the intuitive reading of our iff-definitions which should be a complete definition of its head predicate.

3.2.2 on overall computation

Here we show that $\vdash_{\mathcal{F}}$ preserves equivalence by assuming $G_i's$ to be of the appropriate syntactic form [1] and by using the above results, concerning *constraint rules* and iff-definitions.

split

The case of splitting is rather trivial since the transition, from G_i to G_{i+1}, can in itself been interpreted as a distribution law, and thus equivalence between G_i and G_{i+1} holds directly. The intuitive understanding being that at certain points we wish to create more branches, each one of them, offering some additional definite information.

[1] it can be formally proven that for syntactically correct query and program all consequent derivations preserve the appropriate syntactic form

branch propagate

By branch we refer to the particular (topmost) disjunction to which reasoning is focused. Since by C we refer to a conjunction of constraints at this level we use the term *branch propagate* to refer to the propagation of constraints results of which will affect the whole branch. The syntactic substitution of C *by* C' given that $C \leftrightarrow C'$ guarantees that $G_i \leftrightarrow G_{i+1}$ will also be true in this case.

leaf propagate

A leaf here is to be viewed as a conjunction of constraints participating in a disjunction at a branch level. Here we use distribution laws to show that $[(A \vee B) \wedge C \ \& \ A \wedge C \leftrightarrow B'] \leftrightarrow ((B' \vee B) \wedge C)$, which is then used to derive, $G_i \leftrightarrow G_{i+1}$.

efface & prune

These are ways for discarding inconsistent parts of the search space. With *efface* we discard constraints before we bring them on the outermost level, while *pruning* reasons with constraints at that level. Their common feature is that they both use *falsity* for eliminating disjunctive subformulae. Thus for both $G_i \leftrightarrow G_{i+1}$, follows directly from $\alpha \wedge false \ \leftrightarrow \ false$.

The definition of $\vdash_{\mathcal{F}}$ concerned a number of alternatives. Combined they form the complete definition. Here we have shown that each (and all) of the alternatives characterise an equivalence preserving relation. Thus we can now state that $\vdash_{\mathcal{F}}$ is an equivalence preserving relation when certain properties hold for the constraint rules and iff-definitions.

3.3 The Generalised Framework

Inherited to our definition of the framework's derivability is a certain restriction which may be undesirable for other application areas. Our choice is due to the fact that the restricted framework is more precise and promotes concentration on the important issues (this obviously being dependent to the area under investigation).

The restriction is due to the *unfolding's* characterisation, which requires that the predicate to be unfolded cannot be part of a conjunction participating in some disjunction. Although this has not affected the expressiveness of our restricted framework (ie with respect to the syntax both the general and the restricted version can be used to express the same problems), since we can always *split* and then perform the earlier offending *unfold* operation; nevertheless it imposes a restriction on the structure of our search space (ie the efficiency of certain computations). Since in other areas it could be beneficial (or even of crucial importance) to have arbitrary structures, as representations of the search space, we briefly discuss some of the issues involved in describing the generalised framework.

The key issue here is that we seek to access G_i at an arbitrary level (since we seek to reason about an arbitrary structure). This is cumbersome (if not impossible) when considering the above way for expressing structural requirements on G_i. More specifically it would be cumbersome due to the number of structural possibilities one will have to consider and impossible since there may be an infinite number of such possibilities required.

An alternative route is to use, for the purposes of characterisation, the notion of subformulae rather than the rigid regime described in Section 3.1. Thus if we let $G'_i(F_i)$ denote the fact that F_i is a subformula of G'_i, we can, then, characterise the next step in the computation, by

$$G'_{i+1} = G'_i(F_i/F_{i+1})$$

(meaning the replacement of F_i by F_{i+1} in G'_i). Where $F_i \vdash_{\mathcal{F}} F_{i+1}$ is defined in terms analogous to the one used in the earlier sections (excluding the now irrelevant E_i) .

Although the above extension may seem innocuous enough it still represents a substantial leap in terms of generality. Of course one may start with this general framework and derive our specialised one as a particular instance of it. But then a further unnecessary (to the ends of this investigation) cluster of control rules will have to be introduced.

Extra control will be needed here for defining the relative order of choosing a different F_i over the application of the various derivation rules (ie the amount of time spent in a particular area before moving to another one) as well as for choosing the particular disjunction (area) for the next derivation step. We believe that the expense, paid in the above terms, will be worth paying in cases where local application of constraint rules would be preferable to their global use. Something which is certainly not the case for our approach regarding scheduling problems.

Chapter 4

Control

By control in this dissertation we refer to the decisions/rules which affect the derivation order. These play a central role to our thesis. Particularly so, because we wish to exemplify the easiness and uniformity by which the framework can capture the logic of scheduling applications, while demonstrating how simple changes in the control can affect execution-efficiency. Therefor we intent to provide alternative control strategies which will either map already known algorithms, without loss of efficiency, or will produce yet unexplored ways for solving scheduling problems. These highlight the importance of controlling the application of the above mentioned derivation steps. Which in cases will need to incorporate heuristics for controlling both within-a-step and between-steps decisions.

4.1 Derivation Rules

Apart from any specific rules, some preliminary, general remarks can be made on how to relatively order the derivation steps described in Section 3.1. Starting from the bottom, we have *splitting*, which sets heavy requirements on memory and it is certainly to be assigned lowest priority. On the contrary *pruning* reduces the search space (in a very effective way) thus it should be assigned with topmost priority. Similarly when we *efface*, the search tree is reduced (but less effectively now) thus this takes next priority to *pruning*. Also we may define the relative order of *branch* and *leaf propagation*, by preferring to propagate the more generally applicable constraints (occurring at *branches* over the more local ones. Also since in our approach constraints are the units introducing efficiency, we should defer any *unfolding* until after all propagation and pruning of the search space has been completed.

These result in the following ordering :

pruning
efface

branch propagation
leaf propagation

unfolding
splitting

4.2 Heuristics

As we already have mentioned, heuristics within the framework can be used both for deciding which of the possible next derivations should be followed (between-steps heuristics) and for decisions concerning the particular step (within-a-step heuristics).

As we will latter argue, in more detail, these heuristics reveal a yet unexplored search space for scheduling applications. One may vary the control mechanism in any of the many possible ways and then use the particular settings for specific problems. Such experimentations can lead to algorithms which were previously inaccessible either due to the imperative formulation of the existing ones or due to the inability to explicitly reason about disjunctive constraints (when regarding the existing declarative approaches).

Between-steps heuristics can be used for deciding which of the derivation rules should be applied for the next derivation step. For instance we may demand that *splitting* is always followed by *branch propagation*.

Within-a-step heuristics are concerned with the choice between alternative applications of the same derivation rule. More precisely the choice of different ways of matching G_i for the given rule's syntactic specification. A simple example for such a control rule is to require that all disjuncts of a particular disjunction would be subjected to *leaf propagation* before any other application of the same derivation rule.

Here we should mention that there exist rules which define both the choice for derivation rule and that of a particular matching. For instance, after a *leaf propagation* which implicated *false* an *efface* on the same structure should be the next derivation.

Finally we outline some of the possibilities arising from different control strategies over the *splitting* derivation step. Its special significancy is due to the expectation that reasoning within the framework will result to more intelligence *guesses* as to where and how to *split*. In the general case, after all propagation has been performed, there will still be a number of disjunctive constraints remaining. For each disjunct local propagation will have resulted to a number of interesting measures, upon which we may base an educated guess on which disjunction to *split* next. Such measures may include the number of variable intervals affected (due to local propagation) or the extent to which intervals were *tighten*.

This choice is a non-deterministic task and (the main) responsible for the NP-hard nature of scheduling problems. These metrics provide a new approach on how to use heuristics. The resulting search spaces are yet unexplored although their importance is highlighted by this report. Our argument, that they are an effective means, is based on two considerations. First, that through such heuristics we are capable of using knowledge about the global structure of a certain problem without having to spend an exceedingly large slice of our computation time. While on the other axis we spread the choices which allow from problem independent down to problem specific heuristics (the latter of which should not be confused with the currently available heuristics but should be correlated to taking advantage of the particular constraints, involved in solving specific problems).

Chapter 5

A Job-Shop Example

Here we present a simple job-shop problem. The particular formulation and processing values are taken from the introductory example given by French ([Fre82]). The representation, in terms of iff-definitions and constraint rules and the solution achieved by the framework are then outlined.

5.1 The Problem

Three persons (Adonis, Bert and Caron) share a flat and their newspapers (Times, Guardian, and Sun). Each individual demands to read the newspapers in a certain order (eg Adonis starts with the Times, then reads the Guardian and finishes with the Sun). Also we know that each reader takes a fixed amount of time to read each paper (eg Adonis devotes 60 minutes to reading the Times, 30 minutes for the Guardian and 3 minutes for the Sun). The problem is to find a schedule (ie an order of readers for each paper) which will enable all three of the readers to go out at the earliest possible (since they also share the car).

the specific information

The RHS of Table 5.1 shows the order in which the readers demand to read the newspapers. On the LHS of the same Table are the times taken by each reader for each of the newspapers.

From the demands table (which in the literature are known as technologi-

	a	b	c
m	60	25	10
g	30	75	15
s	2	3	5

AND

	a	b	c
m	1	3	3
g	2	1	2
s	3	2	1

Table 5.1: Job-shop Data

cal constraints) a number of useful constraints can be extracted (for instance that Adonis cannot start reading the Guardian before he has finished the Times).

This formulation is just a fancy guise for the job-shop problem. Where a reader can be viewed as a *job*, a newspaper as a *processor*, and reading as the execution of a task (fragment of a certain job) on a specific processor.

some notation

In what follows let :

a, **b**, **c** stand for Adonis, Bert and Caron
m, **g**, **s** stand for tiMes, Guardian, Sun
$\mathbf{t_{X,Y}}$ be the starting time of reader Y reading paper X
$\mathbf{l_{X,Y}}$ time needed for reader Y to read paper X
X, Y,... variables ranging over times and natural numbers.

5.2 Framework Representation

iff definitions

order_jobs(R1, R2) iff

length($t_{m,R1}$, L1) $\wedge$ length($t_{m,R2}$, L2) $\wedge$ order_tasks(m, R1, R2,L1,L2)
$\wedge$ length($t_{g,R1}$, L1') $\wedge$ length($t_{g,R2}$, L2') $\wedge$ order_tasks(g, R1, R2,L1',L2')
$\wedge$ length($t_{s,R1}$, L1") $\wedge$ length($t_{s,R2}$,L2") $\wedge$ order_tasks(s, R1, R2,L1",L2")

order_tasks(X, R1, R2,L1, L2) iff

$t_{X,R1} + L1 \leq t_{X,R2} \vee t_{X,R2} + L2 \leq t_{X,R1}$

length(X, Y) iff

(X=$t_{m,a} \wedge Y = 60) \vee (X = t_{g,a} \wedge Y = 30) \vee (X = t_{s,a} \wedge Y = 2)$

(X=$t_{m,b} \wedge Y = 25) \vee (X = t_{g,b} \wedge Y = 75) \vee (X = t_{s,b} \wedge Y = 3)$

(X=$t_{m,c} \wedge Y = 10) \vee (X = t_{g,c} \wedge Y = 15) \vee (X = t_{s,c} \wedge Y = 5)$

bound_end_time(X1, X2, X3, W) iff

length(X1, L1) $\wedge$ length(X2, L2) $\wedge$ length(X3, L3)

$\wedge$ X1+L1$\leq$ W $\wedge$ X2+L2$\leq$ W $\wedge$ X3+L3$\leq$ W

set_up_problem iff

$t_{m,a} + 60 \leq t_{g,a} \wedge t_{g,a} + 30 \leq t_{s,a}$

$\wedge\ t_{g,b} + 75 \leq t_{s,b}\ \wedge\ t_{s,b} + 3 \leq t_{t,b}$

$\wedge\ t_{s,c} + 5 \leq t_{g,c}\ \wedge\ t_{g,c} + 15 \leq t_{m,a}$

$\wedge\ 0 \leq t_{m,a}\ \wedge\ 0 \leq t_{g,b}\ \wedge\ 0 \leq t_{s,c}$

constraint rules

- $W \leq t_X \wedge t_X + L \leq t_Y \wedge V\ is\ L + W \rightsquigarrow V \leq t_Y$
- $t_Y \leq W \wedge t_X + L \leq t_Y \wedge V\ is\ W - L \rightsquigarrow t_X \leq V$
- $V \leq t_X \wedge t_X \leq W \wedge W < V \rightsquigarrow false$

- $t_X \leq V \wedge t_X \leq W \wedge V < W \hookrightarrow t_X \leq V$
- $V \leq t_X \wedge W \leq t_X \wedge V < W \hookrightarrow W \leq t_X$

the query

Q : set_up-problem $\wedge bound\_end\_time(t_{s,a}, t_{m,b}, t_{m,c}, 127)$

$\wedge order\_jobs(a, b) \wedge order\_jobs(a, c) \wedge order\_jobs(b, c)$

5.3 Following the Execution

unfolding

The user-predicates set_up_problem and bound_end_time contain no disjunctions and their unfolding is followed by the straightforward application of constraint rules. These result to a number of constraints which either state a quantitative relation between two starting times (originated at set_up_problem) or bound the intervals of a specific start time (lower bounds are due to greater-than-zero constraints and their propagation and upper bounds are funneled from the constraints in bound_end_time). The net result is a number of conjoined constrains which are presented in Table 5.2. We term these as global constraints since we know that they must be consistent with all solutions.

$$
\begin{array}{rcccr}
0 & \leq & t_{m,a} & \leq & 35 \\
60 & \leq & t_{g,a} & \leq & 95 \\
90 & \leq & t_{s,a} & \leq & 125 \\
78 & \leq & t_{m,b} & \leq & 102 \\
0 & \leq & t_{g,b} & \leq & 24 \\
75 & \leq & t_{s,b} & \leq & 99 \\
20 & \leq & t_{m,c} & \leq & 117 \\
5 & \leq & t_{g,c} & \leq & 102 \\
0 & \leq & t_{s,c} & \leq & 97
\end{array}
\quad \wedge \quad
\begin{array}{ccccc}
t_{m,a} & + & 60 & \leq & t_{g,a} \\
t_{g,a} & + & 30 & \leq & t_{s,a} \\
t_{g,b} & + & 75 & \leq & t_{s,b} \\
t_{s,b} & + & 3 & \leq & t_{m,b} \\
t_{s,c} & + & 5 & \leq & t_{g,c} \\
t_{g,c} & + & 15 & \leq & t_{m,c}
\end{array}
$$

Table 5.2: Initial Global Constraints

tracing the disjunctive constraints

In what follows let :

[inc] trail a constraint which is inconsistent with the global constraints,

[vac] (short for vacuous) trail a constraint which adds no (further) useful information when added to the global constraints,

underlined constraints are added to global constraints (due to inconsistency of other disjunct),

DX label a newly introduced disjunction, DXY being a reference to the left (Y=a) or the right (Y=b) disjunct of disjunction labeled by DX.

This is a static trace and thus some of the dynamic aspects cannot be captured (eg the reconsideration of some pending disjunctions, after the introduction of further constraints).

$$
\begin{array}{lll}
\text{D1:} & \underline{t_{m,a} + 60 \leq t_{m,b}} & \vee \quad t_{m,b} + 25 \leq t_{m,a} \\
 & \quad \text{[vac]} & \qquad 103 \leq t_{m,a}[inc]
\end{array}
$$

$$
\begin{array}{lll}
\text{D2:} & t_{m,a} + 60 \leq t_{m,c} & \vee \quad t_{m,c} + 10 \leq t_{m,a} \\
 & \quad 60 \leq t_{m,c}[vac] & \qquad 30 \leq t_{m,a} \\
 & & \qquad 90 \leq t_{m,b}[vac] \\
 & & \qquad 90 \leq t_{g,a} \\
 & & \qquad 120 \leq t_{s,a}[vac] \\
 & & \qquad t_{m,c} \leq 25 \\
 & & \qquad t_{g,c} \leq 10 \\
 & & \qquad t_{s,c} \leq 5[vac]
\end{array}
$$

$$
\begin{array}{lll}
\text{D3:} & t_{m,b} + 25 \leq t_{m,c} & \vee \quad t_{m,c} + 10 \leq t_{m,b} \\
 & \quad 103 \leq t_{m,c}[vac] & \qquad t_{m,c} \leq 92 \\
 & \quad t_{m,b} \leq 92 & \qquad t_{g,c} \leq 72 \\
 & \quad t_{m,a} \leq 32[vac] & \qquad t_{s,c} \leq 17[vac] \\
 & \quad t_{s,b} \leq 89 & \\
 & \quad t_{g,b} \leq 14[vac] &
\end{array}
$$

where : D2b $\wedge$ D3a $\dashv_{\mathcal{F}}$ *false*, [since $103 \leq t_{m,c} \leq 25$]

$$
\begin{array}{lll}
\text{D4:} & t_{g,a} + 30 \leq t_{g,b} & \vee \quad \underline{t_{g,b} + 75 \leq t_{g,a}} \\
 & \quad t_{g,a} \leq -6[inc] & \qquad 75 \leq t_{g,a} \\
 & & \qquad 105 \leq t_{s,a}[vac] \\
 & & \qquad t_{g,b} \leq 20[vac]
\end{array}
$$

$$\begin{array}{lll}
\text{D5:}\ t_{g,a} + 30 \leq t_{g,c} & \vee & \underline{t_{g,c} + 15 \leq t_{g,a}} \\
\quad 105 \leq t_{g,c}[inc] & & \quad t_{g,c} \leq 80 \\
 & & \quad t_{s,c} \leq 75[vac]
\end{array}$$

$$\begin{array}{lll}
\text{D6:}\ t_{g,b} + 75 \leq t_{g,c} & \vee & t_{g,c} + 15 \leq t_{g,b} \\
\quad 75 \leq t_{g,c} & & \quad 20 \leq t_{g,b} \\
\quad 90 \leq t_{g,a} & & \quad 95 \leq t_{s,b} \\
\quad 120 \leq t_{s,a}[vac] & & \quad 98 \leq t_{m,b}[vac] \\
\quad t_{g,b} \leq 5[vac] & & \quad 95 \leq t_{g,a} \\
 & & \quad 125 \leq t_{s,a}[vac] \\
 & & \quad t_{g,c} \leq 5 \\
 & & \quad t_{s,c} \leq 0[vac]
\end{array}$$

$$\begin{array}{lll}
\text{where :} & \text{D6a} \wedge \text{D2b} \dashv_{\mathcal{F}} \textit{false}, & [\text{since } 75 \leq t_{g,c} \leq 10] \\
\text{and :} & \text{D6b} \wedge \text{D3a} \dashv_{\mathcal{F}} \textit{false}, & [\text{since } 98 \leq t_{m,b} \leq 92]
\end{array}$$

$$\begin{array}{lll}
\text{D7:}\ t_{s,a} + 2 \leq t_{s,b} & \vee & \underline{t_{s,b} + 3 \leq t_{s,a}} \\
\quad 92 \leq t_{s,b} & & \quad [\text{vac}] \\
\quad 95 \leq t_{m,b}[vac] & & \\
\quad t_{s,a} \leq 97 & & \\
\quad t_{g,a} \leq 67[inc] & &
\end{array}$$

$$\begin{array}{lll}
\text{D8:}\ t_{s,a} + 2 \leq t_{s,c} & \vee & \underline{t_{s,c} + 5 \leq t_{s,a}} \\
\quad 92 \leq t_{s,c} & & \quad [\text{vac}] \\
\quad t_{s,a} \leq 73[inc] & &
\end{array}$$

$$\begin{array}{lll}
\text{D9:}\ t_{s,b} + 3 \leq t_{s,c} & \vee & \underline{t_{s,c} + 5 \leq t_{s,b}} \\
\quad t_{s,b} \leq 72[inc] & & \quad [\text{vac}]
\end{array}$$

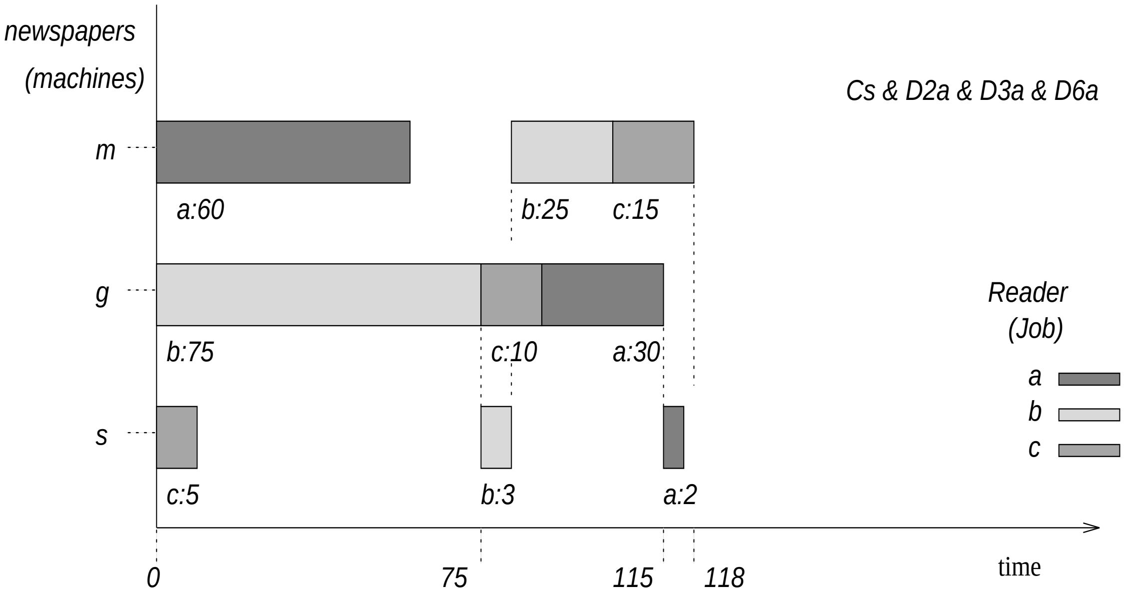

Figure 5.1: Schedule for D2a,D3a,D6a

5.4 The Solution

After the above selection of disjuncts the original conjunction of *order_jobs* predicates has be reduced to :

- Cs [1] $\wedge[(D2 \wedge D3b \wedge D6b) \vee (D2a \wedge D3 \wedge D6a)]$

Which when written in Disjunctive Normal Form (DNF) becomes:

1. $Cs \wedge D2a \wedge D3a \wedge D6a \vee$
2. $Cs \wedge D2a \wedge D3b \wedge D6a \vee$
3. $Cs \wedge D2a \wedge D3b \wedge D6b \vee$
4. $Cs \wedge D2b \wedge D3b \wedge D6b$

Each of the disjuncts being a representation of the schedule in the respective Figure (these being the only schedules achieving execution of less or equal

[1] Cs : $D1a \wedge D4b \wedge D5b \wedge D7b \wedge D8b \wedge D9b$

to 127 time units). The DNF above will be the result of *splitting* the search space when propagation will be completed and no further inconsistencies can be detected. Thus this, DNF, is exactly the output of the framework's computation.

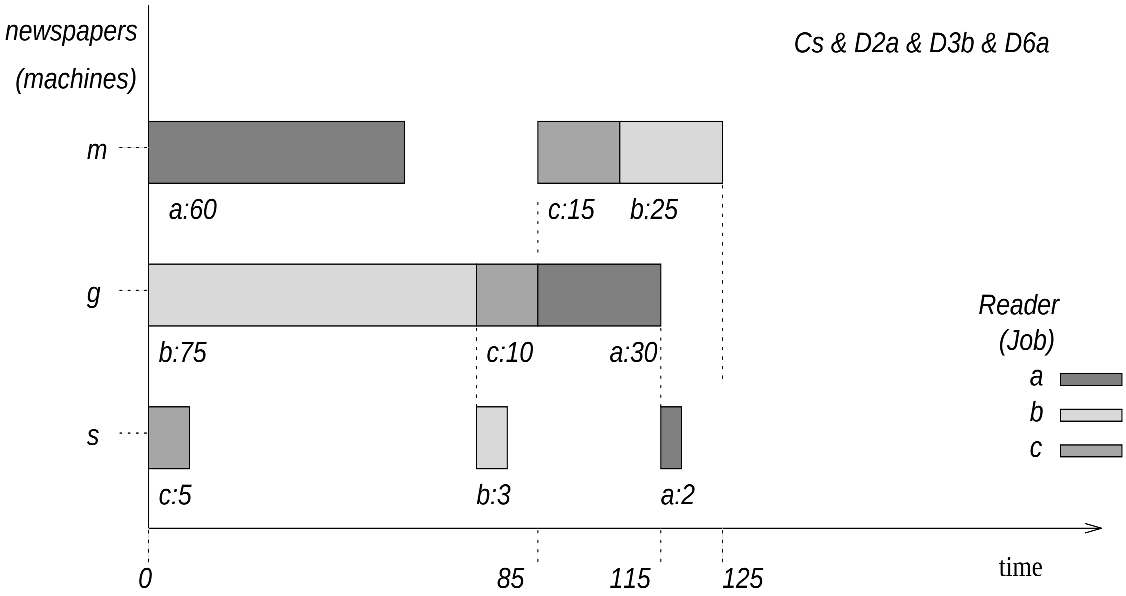

Figure 5.2: Schedule for D2a,D3b,D6a

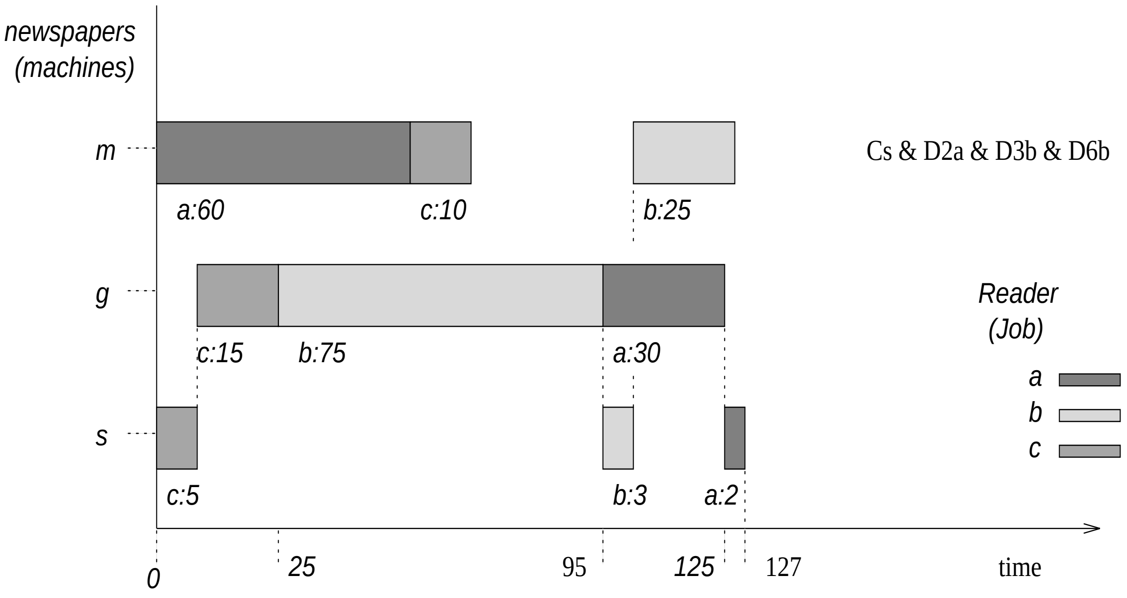

Figure 5.3: Schedule for D2a,D3b,D6b

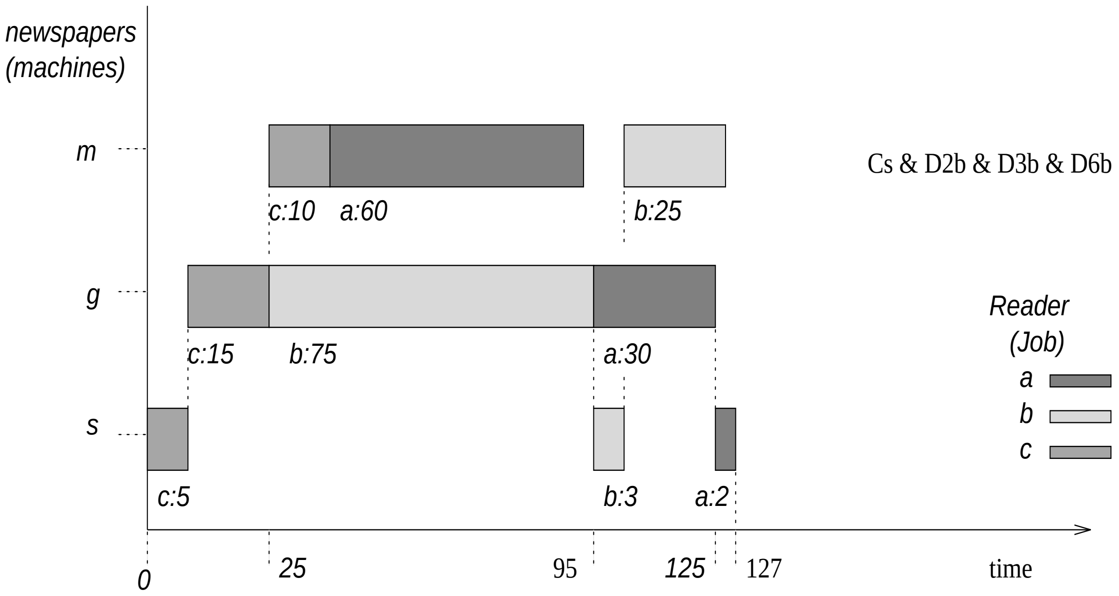

Figure 5.4: Schedule for D2b,D3b,D6b

Chapter 6

Operations Research Techniques

In this chapter we show how the framework can be used to simulate some well established scheduling algorithms; which have been resulted from work in the area of operations research. Such algorithms can be broadly indexed to one of the following categories.

Constructive Algorithms (exact optimality).
Enumeration methods (dynamic programming, branch and bound).
Heuristics (subdivided into)
schedule generation techniques
(eg using Shortest Processing Time) and
more elaborate methods
(eg shifting bottleneck, neighbourhood search etc).

For each of the broad categories, we start by introducing a representative algorithm (usually in the context of some example) and then discuss the logical representation and selection rules needed for simulating the specific algorithm within the framework. Since our primary concern is to demonstrate the flexibility provided by the framework through the various instantiations of selection rules, we keep the logical representation as simple and uniform as possible (this also being true for constraint rules).

6.1 Constructive Algorithms

In this category belong some deterministic algorithms (meaning that no search space of schedules is necessary to be traversed) which only solve special cases of scheduling problems, but for which optimality of the produced schedule can be mathematically established. For such algorithms we claim no particular execution-time gains, from using the proposed framework, but we show the use of the framework as a conceptual tool and the gains in ease of formulation this leads to. The algorithm we chose to investigate here is Johnson's algorithm for the two-machine, n-jobs, flow-shop problem for minimising finishing time ([Joh54]).

6.1.1 Johnson's algorithm

In a two-machine, n-jobs, flow-shop problem all n jobs are composed from two tasks. Furthermore the distinguishing characteristic of flow-shop is that respective tasks across jobs are processed in the same order (without loss of generality we can assume that machine one is the first to process its part of each job). Thus in more concrete terms we have that for every job its task which is operable on the first machine has to be scheduled before the task operable on the second machine. Therefore flow-shop is a special case of the job-shop problems which were introduced in the previous chapter. (Where *flow* refers to the sensation that jobs flow in a common path through the shop and towards completion.)

Johnson's algorithm is based on the observation that for a two-machine setting the time taken for first task scheduled on first machine and last task on the second machine will necessarily introduce some idle time on the complementary machine. Thus one tries to minimise these idle times by finding the most appropriate task for being either first or last in the sequence (depending on which machine the minimum-operation-time task is operable). The algorithm proceeds by selecting one task per cycle, until no job is left unscheduled. For flow-shop problems a schedule can be represented by a sequence of job numbers. This sequence indicates the operation order for both machines. The algorithm guarantees optimality of the the produced schedule

([Joh54]).

Johnson's algorithm (outlined) :

set UnscheduledJobs = Jobs, i=1, j=n
decl array sequence(1..n)
while UnscheduledJobs $\neq \emptyset$ do
    choose $Task_{m,k}$ (having shortest processing time)
        (m:machine, k:job $\in$ UnscheduledJobs)
    If $m = 1$
    then $sequence(i) = k$
        increase i by 1
    else $sequence(j) = k$
        decrease j by 1
    UnscheduledJobs = UnscheduledJobs - Job_k

An illustration of Johnson's algorithm, at work, is shown in Table 6.2 which operates on the data displayed in Table 6.1. Each row represents the contents of *sequence* at successive cycles of the algorithm's execution (starting at the top). On the left partition of the table the selected k is depicted. (Ties are broken in an arbitrary fashion; without loss of optimality.) The solution is then given by the sequence [3, 4, 1, 2], which gives the order in which the jobs (ie their tasks for each machine) should be processed as to minimise finishing time. A pictorial representation is given in Figure 6.1.

Job	M_1	M_2
1	6	4
2	4	3
3	4	5
4	7	6

Table 6.1: Two machine data

_	_	_	_	_
2	_	_	_	2
3	3	_	_	2
1	3	_	1	2
4	3	4	1	2
k	sequence			

Table 6.2: Johnson's solution

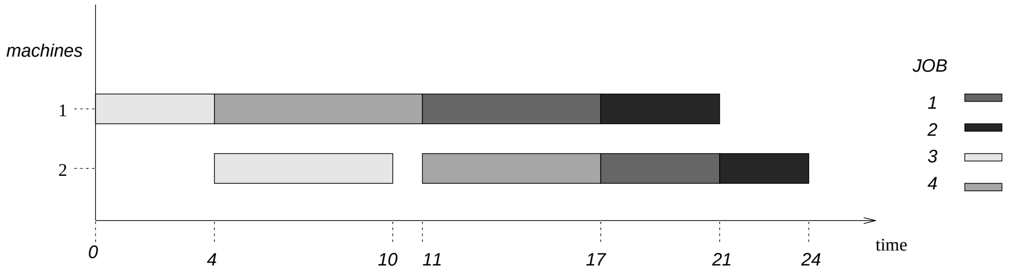

Figure 6.1: Two machine optimal schedule

6.1.2 constraint formulation

Since we are concerned with uniformity on the representation front, here we keep changes of the iff definitions to a possible minimum (as oppose to rethinking the whole of the logical representation). The only difference is due to the special structure of flow-shop problems when seen as special cases of the more general job-shop category of problems. In flow-shop problems optimal solutions have been shown to (always) exist within the class of schedules for which one orders the tasks of a single job in the same order irrespectively of their executing machine. (For instance in Figure 6.1 tasks for job 3 are scheduled first, both for machine 1 and for machine 2; similarly both tasks for job 4 are scheduled second in their respective machine sequence.) Thus here we use the following definition for *order*.

$$
\begin{aligned}
\text{order(X, Y) iff} \quad & t_{1,X} + L1 \leq t_{1,Y} \wedge t_{2,X} + L2 \leq t_{2,Y} \\
& \wedge\, length(t_{1,X}, L1) \wedge length(t_{2,X}, L2) \\
\vee \quad & t_{1,Y} + L1' \leq t_{1,X} \wedge t_{2,Y} + L2' \leq t_{2,X} \\
& \wedge\, length(t_{1,Y}, L1') \wedge length(t_{2,Y}, L2')
\end{aligned}
$$

A further change is that the framework's branch selection rule does not simply choose the branch (ie topmost disjunctive level) to explore first, but rather it deterministicly prefers one (branch) and discards the rest (of the branches). This outstrips the equivalence preserving nature of computation steps (as defined in Section 3.1), but only doing so due to the imperative formulation of the algorithm we wish to simulate. (For this type of algorithms we can claim the desired mathematical properties through the relation to the original mathematical results for the imperative algorithms.)

More specifically, the rule selects the branch involving the lower task time when the particular task is schedulable on the first machine, otherwise it select the disparate disjunct. Following the branch selection (ie having lower precedence, than branch selection) are the *addition* and *deletion* operations. As a matter of avoiding unnecessary work we also introduce the following *deletion* rule $order(X,Y) \wedge (t_{1,X} + Z \leq t_{1,Y} \vee t_{1,Y} + V \leq t_{1,X}) \hookrightarrow order(X,Y)$. The only addition rule used here (since we only are interested in the ordering) is $X + V \leq Y \wedge Y + W \leq Z \wedge UisV + W \rightsquigarrow X + U \leq Z$.

For the query : $order(1,2) \wedge order(1,3) \wedge order(1,4) \wedge$
$order(2,3) \wedge order(2,4) \wedge order(3,4)$

and by letting $\rightsquigarrow, \hookrightarrow$ to respectively precede applications of addition and deletion rules, we get :

$$\underline{t_{1,1} + 6 \leq t_{1,2} \wedge t_{2,1} + 4 \leq t_{1,2}} \quad \vee \quad t_{1,2} + 4 \leq t_{1,1} \wedge t_{2,2} + 3 \leq t_{2,1}$$

where under lined disjuncts are the selected ones, the above selection is based on minimum task-time, which is 3 and since it belongs to a task operable on second machine we choose the dispare disjunct to the one in which it appears.

$$t_{1,1} + 6 \leq t_{1,3} \wedge t_{2,1} + 4 \leq t_{2,3} \quad \vee \quad \underline{t_{1,3} + 4 \leq t_{1,1} \wedge t_{2,3} + 5 \leq t_{2,1}}$$

$$\rightsquigarrow t_{1,3} + 10 \leq t_{1,2} \wedge t_{2,3} + 9 \leq t_{2,2}$$

$$\hookrightarrow \text{order}(\ 2,\ 3)$$

$$t_{1,1} + 6 \leq t_{1,4} \wedge t_{2,1} + 4 \leq t_{2,4} \quad \vee \quad \underline{t_{1,4} + 7 \leq t_{1,1} \wedge t_{2,4} + 6 \leq t_{2,1}}$$

$$\rightsquigarrow t_{1,4} + 13 \leq t_{1,2} \wedge t_{2,4} + 10 \leq t_{2,2}$$

$$\hookrightarrow \text{order}(\ 2,\ 4)$$

$$\underline{t_{1,3} + 4 \leq t_{1,4} \wedge t_{2,3} + 5 \leq t_{2,4}} \quad \vee \quad t_{1,4} + 7 \leq t_{1,3} \wedge t_{2,4} + 6 \leq t_{2,3}$$

From selected constraints we get the same final sequence ([3,4,1,2]) as from Johnson's algorithm. Moreover it can straightforwardly be shown that the particular selection rule and constraint rules always result to the same (or an equivalent one, if ties are broken in distinct ways) sequence as the original algorithm.

6.1.3 relating the logical to the imperative

Our main aim has been to extract the essence of an imperative algorithm and to reformulate it in logical terms. Some people would be ready to argue that doing so, without any loss of efficiency, is a noble task in itself. But further to this we argue that even for such examples as constructive algorithms (and in particular for Johnson's algorithm) which seemingly lay outside the scope of logical frameworks, there are certain gains that can be achieved. In this example the logical formulation along with constraint propagation implement (in essence) a quicksort algorithm. As it probably is obvious to the computer-science inclined reader, this is a most effective way for implementing the pseudo code of Section 6.1.1. Given an appropriate data structure formulation, the problem can be perceived as a sorting problem (having also some extra work to do at the end).

We originally advocated that the particular framework of logic with its enhancement of constraint rules is a useful tool for expressing and solving scheduling problems. Here we show how an imperative algorithm can be simulated (and even improved upon) by using a combination of logic and certain control strategies. With the specific example, emphasising the argument that abstract tools can be used as means of identifying efficiencies and experiment at the conceptual level which can then propagate and take more concrete forms in specialised implementations. This is one of our main claims for the proposed framework, which we believe to provide a practical tool of analysis and experimentation bridging the gap between conceptual problem solving and efficient implementation basis.

6.2 Enumeration Methods

These include such methods as dynamic programming and branch and bound. Their description as enumeration methods derives from the fact that they enumerate (implicitly) the possible schedules, until an optimum has been reached. In what they differ is the way of enumeration. In dynamic programming the main approach is to deterministically go through all the subschedules there exist, building up the partial solutions and only holding the partial solutions that may lead to optimality. Thus it can be said that dynamic programming searches the solutions space in a uniform manner, completely independent of the particular values and structure of a problem. Whereas branch and bound methods search in an unfair fashion, trying to fully explore some branches which will establish some bounds; with the hope that these bounds will be proven useful, in pruning some of the yet unexplored branches.

The relative merits of dynamic programming are that its execution time is predictable, that in the worst case it performs much better than branch and bound and that it is more generally applicable for different objective functions. While, branch and bound achieves much better execution times in the average case, can more readily accommodate the use of heuristics in its search pattern and it can, also, be directed in provide crude approximations which will improve according to the time we wish to spent on solving a particular problem.

Here we concentrate on dynamic programming. Our choice is due to the easiness with which our framework can simulate branch and bound algorithms; as can be seen from exposition in Section 5.2. For a discussion on the relation between dynamic programming and branch and bound techniques the reader is referred to [KS76].

6.2.1 single machine example

In [HK62], the well studied methods of dynamic programming were introduced as a means for solving scheduling problems. As in other areas, where dynamic programming is used, the authors proposed the complete search of sub-spaces. They do so by finding the optimal solution for any m jobs ($1 \leq m \leq n$, where n is number of jobs) and then considering all the possible subschedules of sequencing m+1 jobs. (Here we make no distinction between jobs and tasks since for single machine every job has exactly one task.)

In abstract terms we may characterise the search by :

$$\Gamma(Q) = min_{J_i \in Q}\{\Gamma(Q - \{J_i\}) + \gamma_i(C_Q)\}$$

Where for our purposes Q is a set of jobs, J_i is a single job, $\Gamma(Q)$ is optimal cost when scheduling Q, and $\gamma_i(C_Q)$ is the cost of scheduling J_i last given the Q depending C metric. In more operational terms we could interpret the above as finding the minimum cost for Q by considering how much it would cost to schedule each of the jobs last. The cost of scheduling a job last consists of two parts. One is the $\gamma_i(C_Q)$ part, which measures the particular job's cost at the tail of the sequence. The other being the recursively defined cost of optimally scheduling the rest of the jobs (the $\Gamma(Q - \{J_i\})$ part, which establishes the minimum cost for the problem when last job is excluded). The base case of this recursion can be thought as the zero cost of scheduling zero number of jobs.

Job J_i	J_1	J_2	J_3	J_4
processing time p_i	8	6	10	7
due date d_i	14	9	16	16

Table 6.3: Dynamic data

Q	$\{J_1, J_2, J_3, J_4\}$			
C_Q	31			
last J_i	J_1	J_2	J_3	J_4
$\gamma_i(C_Q)$	$\frac{17}{4}$	$\frac{22}{4}$	$\frac{15}{4}$	$\frac{15}{4}$
$\Gamma(Q - \{J_i\}) + \gamma_i$	6	$\frac{31}{4}$	5	$\frac{23}{4}$
minimum			*	
$\Gamma(Q)$	5			

Table 6.4: Last Iteration

In particular for this example (as presented in [Fre82]) we seek to minimise average tardiness for the problem in Table 6.3. Where by tardiness we refer to the delay of job completions relative to their due dates. This achieved when we use:

$$\gamma_i(C_Q) = \frac{1}{n} max\{0, C_Q - d_i\},\ where\ C_Q = \sum_{i \in Q} p_i\ and\ n = 4$$

In Table 6.4 we present the last iteration of the usual dynamic programming algorithm; in which one starts with Q being empty and so proceeds until Q becomes the set of all jobs. (Which could equally be seen as a recursive procedure, in which case the table depicts the outmost level of recursion, after the return of control from its inward recursion.) A schedule achieving the minimum 5 (time) units of cost is shown in Figure 6.2. In which we see that J_3 is indeed scheduled last as indicated by the asterisk indexing the minimum value in the table presentation.

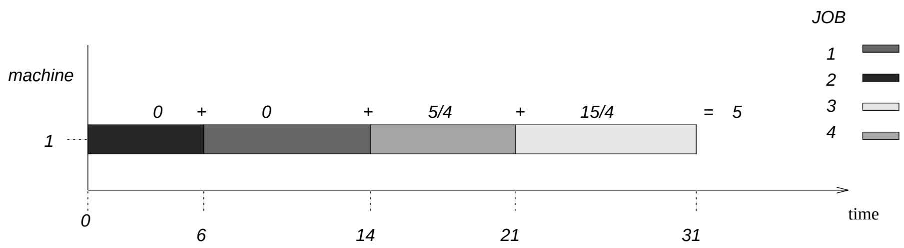

Figure 6.2: Min average tardiness

6.2.2 constraint formulation

Here we need to introduce some new constraints for capturing the notion of a due date. In the inequalities-as-the-only-constraints approach we have followed, up to this point, there exist a strictness on range bounds. Whereas, here, a certain degree of slackness is needed; so that finish dates may exceed their respective due dates. The difference been this of a deadline to a due date. Thus we propose to use a number of global constraints of the form $due(J_i,\ d_i)$ where d_i is the due date for job J_i. This can be introduced similarly to the task length times but play a more active role in the constraint propagation process.

Also we need some mechanism for capturing the incremental nature of the objective function ($\Gamma(Q)$). Since we work within a logical framework we represent Γ by a relation of the form $obj(Q, X)$. Where Q is a set of jobs, and X is the value of $\Gamma(Q)$. Moreover we capture incrementality and dependencies of Γ by considering $obj(Q, X)$ to be a constraint. For the particular example the dependencies may be captured by the following addition rule : $obj(Q,X) \wedge t_I = V \wedge j_I \notin Q \wedge X'isX + max(0, \frac{1}{4}(V + l_i - p_i)) \rightsquigarrow obj(Q \cup \{j_I\}, X')$.

Also there is the need that jobs will be *scheduled* in absolute order (ie first second etc). We can accomplish this by assigning the earliest possible time to the selected job, this is done by : $x \leq t_i \rightsquigarrow x = t_i$.

This *addition* rule violates the requirement set upon *constraint* rules in Section 3.2.1. So since this is not an implication we are unable (as in the constructive algorithms case) to guarantee equivalence. (Note that the first *addition* can be proven to be an implication in the theory of dynamic programming.)

control

The way in which we set the dynamic programming simulation is by requiring splitting of disjunctions to occur in a specific way and by deleting subsumed branches. Where a subsumed branch is one for which the objective function's value is larger to that of another branch while both values are based on same subset of jobs. Splitting has to occur in such a manner so that branches will represent scheduling of tasks to absolute positions. (For example after the first splitting, one branch should be a representation of the search tree in which job j_i is scheduled first and also this should be the case for all i's.) While subsumption is detected by comparing values of *obj* constraints, across branches. (For instance the branch in which j_i is scheduled first and j_k second, may lead to an inferior solution compared to that achieved by scheduling the tasks in the reverse order).

In particular we require that each splitting will represent all the possibilities for scheduling one more job to an absolute position (ie first splitting creates all possibilities for first job, second splitting branches out to schedule the second one and so on). This is done by creating, for each different branch, all the possible selections from pending disjunctions. In each of the spawned branch's all disjunctions in which a certain start time appears will be consistently chosen so that this start time comes before other possibilities.

So for each of these starting times, a branch will be created in which its participating disjunctions will be replaced by the single constraints in which the particular time comes before other starting times. Disjunctions not involving constraints of the particular starting time are unaffected. In more illustrative terms we have :

$$\begin{array}{ll} \text{from} & (t_1 + l_1 \leq t_2 \vee t_2 + l_2 \leq t_1) \wedge (t_1 + l_1 \leq t_3 \vee t_3 + l_3 \leq t_1) \\ & \qquad \wedge\, (t_2 + l_2 \leq t_3 \vee t_3 + l_3 \leq t_2) \\ & \leftrightarrow \\ \text{we get} & \quad\ [t_1 + l_1 \leq t_2 \wedge t_1 + l_1 \leq t_3 \wedge (t_2 + l_2 \leq t_3 \vee t_3 + l_3 \leq t_2)] \\ & \vee\ [t_2 + l_2 \leq t_1 \wedge t_2 + l_2 \leq t_3 \wedge (t_1 + l_1 \leq t_3 \vee t_3 + l_3 \leq t_1)] \\ & \vee\ [(t_3 + l_3 \leq t_1 \wedge t_3 + l_3 \leq t_2 \wedge t_1 + l_1 \leq t_2 \vee t_2 + l_2 \leq t_1)] \end{array}$$

Where the uppermost part of the equivalence is the pre-splitting search space; having the declarative reading of ordering the three tasks. From this we seek to get (by splitting, marked by $\leftrightarrow$ above) a specific reshuffle (while preserving equivalence). The scenarios we are interested in are the ones in which one of the tasks is scheduled to be the first. When considering all tasks in such manner then we have a complete description of the search space. Thus the three lines of the bottom part of the equivalence represent schedules in which either of task 1, task 2 or task 3 (from top to bottom) is scheduled to be first in the schedule represented by the outermost disjuncts.

Following a *splitting* step there is some constraint propagation for each branch. Firstly a single start-time is *bound* to its lower permitted value (the same starting time as the one used for splitting the particular branch). Followed by propagation based on the addition of implied inequalities and on the addition rule involving the *obj* constraint.

Such an approach, of generating more and more branches (by means of distributing conjunctions over disjunctions) leads to an exponential need for memory. So it is vital, for computational purposes, to eliminate branches as soon as their non-optimality can be established. Thus we seek a control rule which will only dispose non-optimal branches while it will do so on a large enough number of branches as to be effective. A further concern is for the rule to have a mathematical theory on which we can show equivalence of computation steps (this quite different from the similarly termed issue on the introduced *addition* rule). Our proposal is to use the *obj* constraint. Thus, if $G_i = A \vee \ldots \vee B$, and $DT \wedge A \vdash obj(S,V)$ and $DT \wedge B \vdash obj(S,W)$ then we delete A or B depending whether $V > W$ is true or not.

The desired mathematical theory can be easily reached in the work already done on proving the optimality of attained schedules for dynamic programming. All that is needed is to map each branch of our representation to a single branch in the problems search space and then use the mathematician's reasoning for pruning and selecting values. Also for the purposes of complexity comparison we can use this mapping to illustrate that our simulation only examines the branches (of search space) examined by the dynamic programming formulation.

In what follows let

$S_{i,k,\ldots}$	stand	for	the Set $\{J_i, J_k, \ldots\}$
$D_{i,k}$	stand	for	$t_i + l_i \leq t_k \vee t_k + l_k \leq t_i$
$C_{i,k,\ldots}$		for	$t_i + l_i \leq t_k \wedge \ldots \wedge t_k + l_k \leq t_j \wedge \ldots$
			ie where i is scheduled[1]first, k second, ...
$\leftrightarrow$	preceding		a *splitting* step,
$\rightsquigarrow$	preceding		x applications of *addition* rules,
$\not\rightarrow$	preceding		a deleted branch,
$obj(\emptyset, 0)$ and $0 \leq t_x$		be	global constraints.

Using this notation the operational behaviour of the instantiated framework for the above one-machine example can be depicted by :

$$D_{1,2} \wedge D_{1,3} \wedge D_{1,4} \wedge D_{2,3} \wedge D_{2,4} \wedge D_{3,4}$$

$$\begin{array}{lll}
\leftrightarrow & & C_1 \wedge D_{2,3} \wedge D_{2,4} \wedge D_{3,4} \\
& \vee & C_2 \wedge D_{1,3} \wedge D_{1,4} \wedge D_{3,4} \\
& \vee & C_3 \wedge D_{1,2} \wedge D_{1,4} \wedge D_{2,4} \\
& \vee & C_4 \wedge D_{1,2} \wedge D_{1,3} \wedge D_{2,3}
\end{array}$$

[1]eg $C_{1,2} = t_1 + l_1 \leq t_2 \wedge t_1 + l_1 \leq t_3 \wedge t_1 + l_1 \leq t_4 \wedge t_2 + l_2 \leq t_3 \wedge t_2 + l_2 \leq t_4$

$$
\begin{array}{lll}
\rightsquigarrow & & obj(S_1,0) \wedge C_1 \wedge D_{2,3} \wedge D_{2,4} \wedge D_{3,4} \\
& & \quad \wedge\, t_1 = 0 \wedge 8 \leq t_2 \wedge 8 \leq t_3 \wedge 8 \leq t_4 \\
& \vee & obj(S_2,0) \wedge C_2 \wedge D_{1,3} \wedge D_{1,4} \wedge D_{3,4} \\
& & \quad \wedge\, t_2 = 0 \wedge 6 \leq t_1 \wedge 6 \leq t_3 \wedge 6 \leq t_4 \\
& \vee & obj(S_3,0) \wedge C_3 \wedge D_{1,2} \wedge D_{1,4} \wedge D_{2,4} \\
& & \quad \wedge\, t_3 = 0 \wedge 10 \leq t_1 \wedge 10 \leq t_2 \wedge 10 \leq t_4 \\
& \vee & obj(S_4,0) \wedge C_4 \wedge D_{1,2} \wedge D_{1,3} \wedge D_{2,3} \\
& & \quad \wedge\, t_4 = 0 \wedge 7 \leq t_1 \wedge 7 \leq t_2 \wedge 7 \leq t_3
\end{array}
$$

$$
\begin{array}{lll}
\leftrightarrow, \rightsquigarrow & & obj(S_{1,2},\frac{5}{4}) \wedge C_{1,2} \wedge D_{3,4} \\
& & \quad \wedge\, t_1 = 0 \wedge t_2 = 8 \wedge 14 \leq t_3 \wedge 14 \leq t_4 \\
& \vee & obj(S_{1,2},0) \wedge C_{2,1} \wedge D_{3,4} \\
& & \quad \wedge\, t_2 = 0 \wedge t_1 = 6 \wedge 14 \leq t_3 \wedge 14 \leq t_4 \\
& \vee & obj(S_{1,3},\frac{1}{2}) \wedge C_{1,3} \wedge D_{2,4} \\
& & \quad \wedge\, t_1 = 0 \wedge t_3 = 8 \wedge 18 \leq t_2 \wedge 18 \leq t_4 \\
& \vee & \ldots \\
& \vee & obj(S_{1,3},1) \wedge C_{3,1} \wedge D_{2,4} \\
& & \quad \wedge\, t_3 = 0 \wedge t_1 = 10 \wedge 18 \leq t_2 \wedge 18 \leq t_4 \\
& \vee & \ldots
\end{array}
$$

$$
\begin{array}{ll}
\not\rightarrow & obj(S_{1,2},\frac{5}{4}) \wedge C_{1,2} \ldots \\
& obj(S_{1,3},1) \wedge C_{3,1} \wedge D_{2,4} \ldots \\
& obj(S_{1,4},1) \wedge C_{4,1} \wedge D_{2,3} \ldots \\
& obj(S_{2,3},\frac{7}{4}) \wedge C_{3,2} \wedge D_{1,4} \ldots \\
& obj(S_{2,4},1) \wedge C_{4,2} \wedge D_{1,3} \ldots \\
& obj(S_{3,4},1) \wedge C_{3,4} \wedge D_{1,2} \ldots
\end{array}
$$

Each of the above branches is deleted because somewhere in the search space there exist an *obj* function, which relates the same set of jobs to a smaller value.

so the remaining branches are :

$$
\begin{array}{l}
obj(S_{1,2},0) \wedge C_{2,1} \wedge D_{3,4} \\
\quad \wedge\, t_2 = 0 \wedge t_1 = 6 \wedge 14 \leq t_3 \wedge 14 \leq t_4
\end{array}
$$

$$
\begin{array}{ll}
 & obj(S_{1,3}, \frac{1}{2}) \wedge C_{1,3} \wedge D_{2,4} \\
 & \quad \wedge\, t_1 = 0 \wedge t_3 = 8 \wedge 18 \leq t_3 \wedge 18 \leq t_4 \\
 & obj(S_{1,4}, 0) \wedge C_{1,4} \wedge D_{2,3} \\
 & \quad \wedge\, t_1 = 0 \wedge t_4 = 8 \wedge 15 \leq t_2 \wedge 15 \leq t_3 \\
 & obj(S_{2,3}, 0) \wedge C_{2,3} \wedge D_{1,4} \\
 & \quad \wedge\, t_2 = 0 \wedge t_3 = 6 \wedge 16 \leq t_1 \wedge 16 \leq t_4 \\
 & obj(S_{2,4}, 0) \wedge C_{2,4} \wedge D_{1,3} \\
 & \quad \wedge\, t_2 = 0 \wedge t_3 = 6 \wedge 13 \leq t_1 \wedge 13 \leq t_3 \\
 & obj(S_{3,4}, \frac{1}{4}) \wedge C_{4,3} \wedge D_{1,2} \\
 & \quad \wedge\, t_4 = 0 \wedge t_3 = 7 \wedge 17 \leq t_1 \wedge 17 \leq t_3
\end{array}
$$

$$
\begin{array}{ll}
\leftrightarrow, \leadsto, \not\rightarrow & obj(S_{1,2,3}, 2) \wedge C_{2,1,3} \\
 & \quad \wedge\, t_2 = 0 \wedge t_1 = 6 \wedge t_3 = 14 \wedge 24 \leq t_4 \\
 & obj(S_{1,2,4}, \frac{5}{4}) \wedge C_{2,1,4} \\
 & \quad \wedge\, t_2 = 0 \wedge t_1 = 6 \wedge t_3 = 14 \wedge 21 \leq t_3 \\
 & obj(S_{1,3,4}, \frac{9}{4}) \wedge C_{1,4,3} \\
 & \quad \wedge\, t_1 = 0 \wedge t_4 = 8 \wedge t_3 = 15 \wedge 25 \leq t_2 \\
 & obj(S_{2,3,4}, \frac{7}{4}) \wedge C_{1,4,3} \\
 & \quad \wedge\, t_1 = 0 \wedge t_4 = 8 \wedge t_3 = 15 \wedge 23 \leq t_2
\end{array}
$$

$$
\begin{array}{ll}
\leadsto, \not\rightarrow & obj(S_{1,2,3,4}, 5) \wedge C_{2,1,4,3} \\
 & \quad \wedge\, t_2 = 0 \wedge t_1 = 6 \wedge t_4 = 14 \wedge t_3 = 21
\end{array}
$$

6.2.3 relating the logical to the imperative

Here we have shown how a certain class of algorithms can be decomposed into two parts. One part is concerned with their logic and the other with their control. Our approach has been inspired by [Kow79] were the particular way of decomposing algorithms was first introduced. Here the use of constraints rules can be viewed as a handle provided to the user towards the abstract definition of control.

Although we present no formal complexity analysis, it is clear that the above presentation creates a one-to-one mapping between the branches explored by the instantiated framework and those explored by dynamic programming algorithms. Furthermore the deletion of branches is the counterpart of choosing the optimal sub-schedule; although less efficient in our general treatment due to its ample generality. Similarly the calculation of C_Q is mapped to the process of propagating lower bounds on starting times and its associative binding to a single value. Again this may be marginally less efficient due to the propagation of lower bounds in branches, which in our approach is carried to all the starting times irrespectively of the branch's fate. Both of the above differences can be shown to affect only the constants of the complexity formula or even that they can be eliminated, if one cares to define the control structure to a finer degree (something which outside the scope of this section.)

Our main claim here is that the proposed framework improves on the expressiveness of present logic-based computational frameworks. Readers who are familiar with Logic Programming will surely appreciate the power we are granted by the possibility of reasoning over disjunctive constraints-predicates. Power for which we normally have to pay with the degradation of execution-efficiency. In our attempt to harness this we have replaced backtracking by constraint satisfaction.

In this example the logic representation was unaltered and the control was defined in a clear, precise way (following the logic of the dynamic programming paradigm). Trying to capture the disjunctive nature of constraints, in scheduling applications, within the normal Logic Programming is far from trivial. Far from trivial is also their treatment in constraint based approaches (eg [VH89]).

6.3 Heuristic Methods

In the classes of operations research algorithms we have considered in the preceding sections we were guaranteed to find an optimal schedule (and they can be trivially enhanced as to find all of the optimal schedules). The shortcomings of limited applicability for the constructive case and degraded performance for dynamic programming, can only be overcome when one sacrifices some of the confidence in that the found schedule will be an optimal one. Many different approaches have been proposed in this area (which is very well researched due to its pertinence to practical applications of scheduling). These include generation of active schedules in [GT60] and its variant on generating non active schedules, neighbourhood search in [Bak74] with a recent approach in [SWV92] and the shifting bottleneck procedure in [ABZ88].

Here we concentrate on the generation of non-delay schedules using priority rules. A non-delay schedule is one in which no machine remains inactive when there exists a task which could be executed on the particular machine; given that no violation of precedence constraints occurs. This class is a subclass of the active schedules class, which requires that a task cannot be scheduled any earlier without delaying some other task. Finally a superclass for the active class is the class of semi-active schedules; which requires that for a specific processing sequence, there does not exist any task which could be scheduled at an earlier time without changing this sequence. This classification can be seen in Figure 6.3.

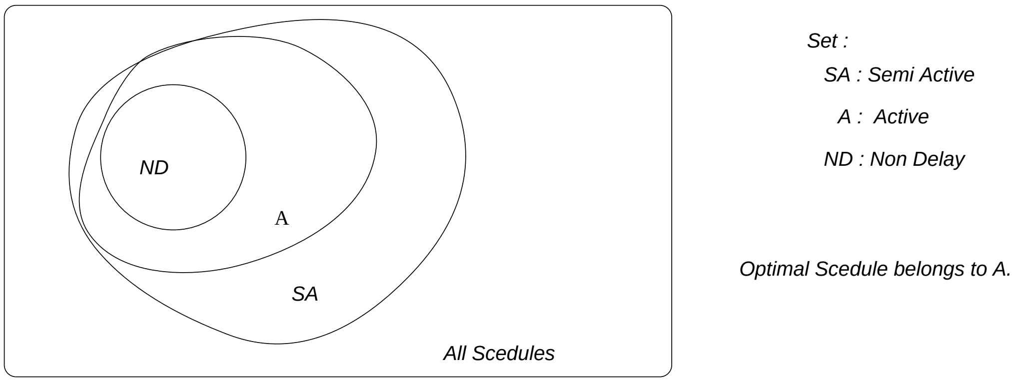

Figure 6.3: A Classification of Job-Shop Schedules.

6.3.1 non-delay with priority rules

When generating the whole space for some of the above classes of schedules or at least implicitly enumerating the members of such a class, we run into problems, which are similar to the ones encountered in the dynamic programming case. (Which is an implicit enumeration of the active class, the class which is guaranteed to contain the optimal schedule; while also being the smaller of the three for which this is always the case.) To overcome these problems one may use heuristics for choosing only one of a number of possible choices at certain points of computation. One such approach uses priority rules to choose deterministically one of the possible choices (of which task should be scheduled next) while ignoring the other options (thus trading optimality for efficiency).

Priority rules include :

SPT	Shortest processing time,
FCFS	First Come First Served, order of becoming schedulable, determines priority
MWR	Most Work Remaining, prefer task from job with greatest total remaining, processing time
MOPNR	Most Operations Remaining, choose from job with most unscheduled tasks.

The general algorithm for generating non-delay schedules is given in Table 6.5. Further instantiation follows the selection of some priority rule(s) (such as the above) for the choice in step 3.

Step	
1	**let** $S = \{j_{x,y} \vert j_{x,y}$ has no predecessors$\}$, P=$\emptyset$
	while $S \neq \emptyset$ **do**
2	*find* $\sigma = min_{\sigma_{\mu,k}} \{\sigma_{\mu,k} \vert j_{\mu,k} \in S, \sigma_{\mu,k} = earliest\ possible\ for\ j_{\mu,k}\}$ *where* μ = the associated machine
3	*choose* i such that $j_{\mu,i} \in S$ & $\sigma_{\mu,i} = \sigma$
4	**let** P = P $\cup$ $\{j_{\mu,i}\}$, S = S - $\{j_{\mu,i}\}$
5	S = S $\cup$ {job-successor($j_{\mu,i}$)}

Table 6.5: Generation of non-delay schedules

The essence of the algorithm is that at each iteration we choose the task associated to the earliest starting time of all pending tasks. A pending task is a yet unscheduled task, for which all predecessors have already been scheduled. Its starting time is constrainted by both the finishing times of its job predecessor and its machine predecessor. (The greater of which is its possible starting time.) Priority rule(s) are used to choose between pending tasks which are to be scheduled on the same machine and have identical possible starting times.

In particular at Step 2 of the algorithm we find the earliest time, at which any of the pending tasks may be scheduled (ie selecting the smallest *possible* time). Step 3 uses the choice priority rule for selecting between tasks that are schedulable on the same machine and with the same starting time. Step 4 simply updates the sets of jobs, S and P, and Step 5 adds tasks dependent on the scheduled task to the set of schedulable tasks (S).

An example of this algorithm at work, for the problem with data given in Table 6.7 and where SPT was used, is given in Table 6.6. The data tables should be read as having machines representing the rows and jobs for the columns. Entries in the uppermost table show duration of tasks (eg the task of first job which is executable on second machine has a duration of 14 units), while at the bottom we show the order of tasks within jobs (eg task of second job which is executable on third machine must be scheduled after task, of the same job, executable on second machine.

Where the priority rule is used for choosing one of the elements from the set appearing on the fourth column of the table.

The cautionary points are two. First that non-delay schedules do not in general circumscribe the optimal solution. Second that in such deterministic reduction of the search space, even the local optimum may be disavowed. (In the particular example it would be more profitable to schedule $t_{3,2}$ before $t_{3,1}$; contrary to what our choice of priority rule indicates.)

6.3.2 constraint formulation

Since for this algorithm there is only a very limited number of choice points, it would lead to a less direct mapping if we were to insist in the disjunctive formulation. Rather we will concentrate our attention to the constraint rules. Aiming to show their expressive and computational potentials.

S	σ	μ	$\{ t_{\mu,x}\text{'s} \}$	$t_{\mu,j}$
$\{ t_{1,1}, t_{1,2}, t_{3,3} \}$	0	1	$\{ 1,2 \}$	$t_{1,1} = 0$
$\{ t_{2,1}, t_{1,2}, t_{3,3} \}$	0	3	$\{ 3 \}$	$t_{3,3} = 0$
$\{ t_{2,1}, t_{1,2}, t_{1,3} \}$	10	1	$\{ 2 \}$	$t_{1,2} = 10$
$\{ t_{2,1}, t_{3,2}, t_{1,3} \}$	10	2	$\{ 1 \}$	$t_{2,1} = 10$
$\{ t_{3,1}, t_{3,2}, t_{1,3} \}$	24	1	$\{ 3 \}$	$t_{1,3} = 24$
$\{ t_{3,1}, t_{3,2}, t_{2,3} \}$	24	3	$\{ 1,2 \}$	$t_{3,1} = 24$
$\{ t_{3,2}, t_{2,3} \}$	26	2	$\{ 3 \}$	$t_{2,3} = 26$
$\{ t_{3,2} \}$	28	3	$\{ 2 \}$	$t_{3,2} = 28$
$\{ t_{2,2} \}$	43	2	$\{ 2 \}$	$t_{2,2} = 43$

Table 6.6: Non-delay schedule using SPT

	1	2	3
1	10	14	2
2	14	3	8
3	4	15	12

&

	1	2	3
1	1	1	2
2	2	3	3
3	3	2	1

Table 6.7: NonDelay data

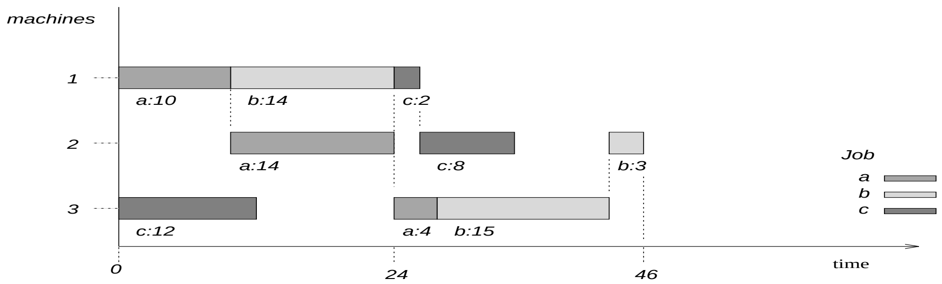

Figure 6.4: The produced non delay schedule

Most of the calculations in the above example are embraced by the already-introduced *constraint rules*. For example the starting times and predecessor-successor notions. The crux for a complete mapping is to be found in the representation of priority rule. In the general case we need some propagation rules which will function as a local objective function, according to the conjunctive constraints at hand. Furthermore some control will be normally needed for guiding the application of such rules.

For the particular priority rule (SPT) the *addition* constraint rule has only to consider two conflicting tasks. As in earlier examples we represent the logic of a relative ordering which then, via propagation steps delivers a total ordering.

Here we have :

1 The generally available constraint rules (as in Section 5.2)

2 $length(t_{x,y}, V) \wedge length(t_{x,z}, W) \wedge V < W \wedge$

$$\phi \leq t_{x,y} \wedge \phi \leq t_{x,z} \rightsquigarrow t_{x,y} + l_{x,y} \leq t_{x,z}$$

(relative ordering of two competing tasks)

3 $\phi \leq t_{x,y} \rightsquigarrow \phi = t_{x,y}$

(possible start time becomes actual start time)

4 $\phi \leq t_{x,y} \wedge \phi = t_{x,y} \hookrightarrow \phi \leq t_{x,y}$

(possible start time is now redundant)

The last requirement for simulating the generation of non-delay schedules is set upon control. So in order to avoid premature (and probably incorrect) application of the third rule we demand that, first, all propagation involving rules in 1 and 2 will be attempted before any propagations involving rules 3 and 4 (obviously in the given order) and second that ϕ is the minimum value appearing in such a constraint.

Following the general exposition of Section 5.2 we can directly define the program which is essentially a grouping for constraints. So we start our tracing of the example from the original set of constraints and follow this by the annotated transitions. (For clarity's shake we, throughout, assume the presence of the appropriate length constraints.) The resulting schedule is that of Figure 6.4.

$$\begin{array}{lclcl}
t_{1,1} + 10 \leq t_{2,1} & & t_{1,2} + 14 \leq t_{3,2} & & t_{3,3} + 12 \leq t_{1,3} \\
t_{2,1} + 14 \leq t_{3,1} & \bigwedge & t_{3,2} + 15 \leq t_{2,2} & \bigwedge & t_{1,3} + 2 \leq t_{2,3} \\
0 \leq t_{1,1} & & 0 \leq t_{1,2} & & 0 \leq t_{3,3}
\end{array}$$

$$\begin{array}{ll}
2 \rightsquigarrow & t_{1,1} + 10 \leq t_{1,2} \\
3 \overset{2}{\rightsquigarrow} & t_{1,1} = 0, t_{3,3} = 0 \\
1 & (t_{1,1} + 10 \leq t_{2,1} \leftrightarrow 10 \leq t_{2,1}), \\
 & \quad (t_{3,3} + 12 \leq t_{1,3} \leftrightarrow 12 \leq t_{1,3}) \\
 & \quad (t_{1,1} + 10 \leq t_{1,2} \leftrightarrow 10 \leq t_{1,2}) \\
3 \overset{2}{\rightsquigarrow} & t_{2,1} = 10, t_{1,2} = 10 \\
1 & (t_{2,1} + 14 \leq t_{3,1} \leftrightarrow 24 \leq t_{3,1}),
\end{array}$$

$$
\begin{array}{lll}
 & & (t_{1,2} + 14 \leq t_{3,2} \leftrightarrow 24 \leq t_{3,2}) \\
2 & \rightsquigarrow & t_{3,1} + 4 \leq t_{3,2} \\
3 & \stackrel{2}{\rightsquigarrow} & t_{1,3} = 24, t_{3,1} = 24 \\
1 & & (t_{1,3} + 2 \leq t_{2,3} \leftrightarrow 26 \leq t_{2,3}), \\
 & & (t_{3,1} + 4 \leq t_{3,2} \leftrightarrow 28 \leq t_{3,2}) \\
3 & \rightsquigarrow & t_{2,3} = 26 \\
3 & \rightsquigarrow & t_{3,2} = 28 \\
1 & & (t_{3,2} + 15 \leq t_{2,2} \leftrightarrow 43 \leq t_{2,2}) \\
3 & \rightsquigarrow & t_{3,2} = 43
\end{array}
$$

$$
\begin{array}{lclcl}
t_{1,1} = 0 & & t_{1,2} = 10 & & t_{1,3} = 24 \\
t_{2,1} = 10 & \bigwedge & t_{2,2} = 43 & \bigwedge & t_{2,3} = 26 \\
t_{3,1} = 24 & & t_{3,2} = 28 & & t_{3,3} = 0
\end{array}
$$

6.3.3 relating the logical to the imperative

Clearly in this example the strongest of the framework's potentials remained unexploited (namely the use of disjunctions). Instead the computation load was focused in using constraint rules; thus demonstrating the flexibility and expressiveness the user gains from having the ability to define her own rules. This is in contrast with the mainstream approaches to integrating constraints in logic environments, such as [DHS+88] and [JL87].

The main computational benefit derived from applying the framework to this example can be realised when observing the application of the third propagation rule. One can exploit the independency of tasks starting at the same time but are to be scheduled on different machines, by scheduling them in a single iteration (whereas the imperative formulation would require multiple iterations; one for each task). Of course this is a deficiency of the particular imperative algorithm which can be rectified. This being in exact armony with one of the uses which we argue that the framework is particularly suitable for (ie a tool for analysis and explorative work, through which a better understanding can be gained and the exact parameters of a solution may be determined).

When comparing the two algorithms, from a strictly performance-based view of efficiency, we only have one major difference to consider. This being the way of choosing according to the priority rule. It may seem that the framework's approach may be rather wasteful when more then two choices are presented to the priority rule. This is due to simplicity being our aim for the particular formulation. Also our restrictions on control have been kept to a bare minimum. Thus to reclaim efficiency on the particular issue we can either employ a monotonic function similarly to *obj* used in Section 6.2.3 or alternatively define the control structure to a finer degree.

Chapter 7

Conclusions

This thesis had two primary aims. First to introduce a new logical framework and second to heft the framework's applicability to a certain problem domain.

As with respect to our first aim, the main contributions is through the formal exposition, for which although we do not claim any degree of perfection, it still, successfully identifies the different technologies and a route via which they can be brought together. The other, complementary to the above, main contribution of this work at the conceptual level is that of awareness, awareness on the issues involved when we reason over disjunctive information (awareness indeed that what we really do is, in essence, to reason about disjunctive constraints).

Coming to the issues involved in efficient, scheduling-related, problem solving this dissertation has to offer rather little. Our principal, practical, results had to be limited, since the time (three and a half months) did not allow for an implementation of the proposed framework. Thus we have focused our attention to some illustrative, simple, examples and on simulating already existing techniques. The conclusions one may draw from the simulating examples are encouraging since the framework provided clear ways for mapping the simulated techniques to declarative representations.

This work has made clear the suitability of constraints for replacing inefficient backtracking (over disjunctions of clauses, as dictated by Logic Programming), by forward elimination of inconsistent combinations; particularly so for scheduling applications.

The most controversial of the introduced notions is the generalisation of the way consistency is handled. The points which we have found to support our view are:

- homogeneity of underlying principles (ie of the principles deriving inconsistency)

- extending consistency is the next step to what CLP approaches have been, all along (ie to extend unification by constraint satisfaction)

- it provides a powerful tool to the user, which enables her to have a substantial control over efficiency while concentrating on abstract aspects of the problem.

The main drawback of this approach is that the mathematical properties, which we are able to establish a priori, are quite limited. To compensate this, the framework offers a particularly simple setting for reasoning, in a straightforward manner, about concrete instances of itself; which are the result of choosing specific constraint rules.

7.1 Further Work

This dissertation is to the proposed framework what a feasibility study is to a computer project. We have merely outlined possible areas, from which certain gains may be accomplished. The next obvious step would require an experimental system to be built, so that feedback from real life problems may be obtained. This is also necessary for identifying other areas to which the framework maybe beneficially applied.

In a more abstract level, further work is needed for a formal definition of control and its effects on derivation. The problems arising in this area can be correlated to the problems of formalising don't care nondeterminism of logic-based concurrent systems. The use of meta constraints for such purposes may be a fruitful direction.

Bibliography

[ABZ88] Joseph Adams, Egen Balas, and Daniel Zawack. The shifting bottleneck procedure for job shop scheduling. *Managment Science*, 34(3), March 1988.

[Bak74] K. R Baker. *Introduction to Sequencing and Scheduling*. John Wiley, New York, 1974.

[Col90] A. Colmerauer. An introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[DHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Agoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Inernational Conference on FGCS (FGCS'88)*, Tokyo, Japan, December 1988.

[FH93] Thom Fruehwirth and Philipp Hanschke. Terminological reasoning with constraint handling rules. In *First Workshop on Principles and Practice of Constraint Programming*, Newport, Rhode Island, USA, April 28-30 1993.

[Fre82] Simon French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*. Ellis Horwood, 1982.

[GT60] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operational Research*, 8:487–503, 1960.

[Har90] S. Haridi. A Logic Programming language based on the Andorra model. *New Generation Computing*, 1990.

[Her93] Patrick Herring. The framework applied to planning. Master's thesis, Imperial College, Department of Computing, London, 1993.

[HK62] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *J. SIAM*, 10:196–210, 1962.

[Hog90] Christopher J. Hogger. *Essentials of Logic Programming*. Oxford University Press, 1990.

[JL87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, Germany, January 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *Fourth International Conference on Logic Programming (ICLP-91)*, Melbourne, Australia, May 1987.

[Joh54] S. M. Johnson. Optimal two- and tree-stage production schedules with set-up times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954.

[Kow79] Robert A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–435, July 1979.

[Kri84] F. Kriwaczek. A critical path analysis program. In C. Clark and F. McCabe, editors, *Micro-Prolog: Programming in Logic*, pages 277–299. Prentice Hall, 1984.

[KS76] W. H. Kohler and K. Steiglitz. Enumerative and iterative computational approaches. In E. G. Coffman, editor, *Computer and Job-Shop Scheduling Theory*, chapter 6, pages 278–286. John Wiley & Sons, 1976.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, March 1977.

[SH92] Greg Sidebottom and William S. Havens. Hierharchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):31–35, 1992.

[SWV92] R. H. Storer, S. D. Wu, and Rendo Vaccari. New search spaces for sequencing problems with application to job-shop scheduling. *Managment Science*, 38(10):1495–1509, October 1992.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[VHD91] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A new logical connective and its application to Constraint Logic Programming. In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris, France, June 1991.

[Wal93] M. Wallace. Applying constraints for scheduling. In *NATO Advanced Study Institute on Constraint Programming*, Parnu, Estonia, August 1993. Springer Verlag. to appear.