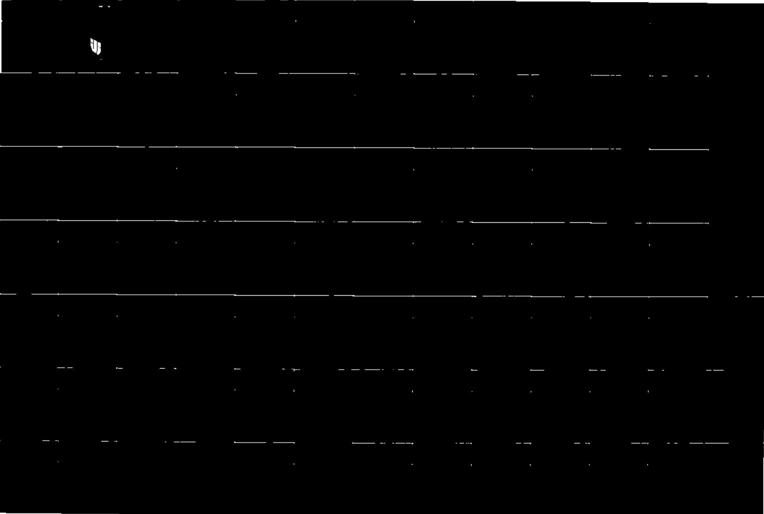
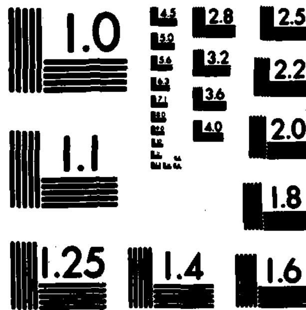
AD-A171 514 INTELLIGENT USE OF CONSIRAINTS FOR ACTIVITY SCHEDULING 1/2
(U) CONSTRUCTION ENGINEERING RESEARCH LAB (ARMY)
CHAMPAIGN IL NAVINCHANDRA AUG 86 CERL-TM-N-86/15

UNCLASSIFIED F/G 12/2 NL





:CROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

US Army Corps
of Engineers
Construction Engineering
Research Laboratory

AD-A171 514

USA-CERL

TECHNICAL MANUSCRIPT N-86/15
July 1986

Intelligent Use of Constraints for Activity Scheduling

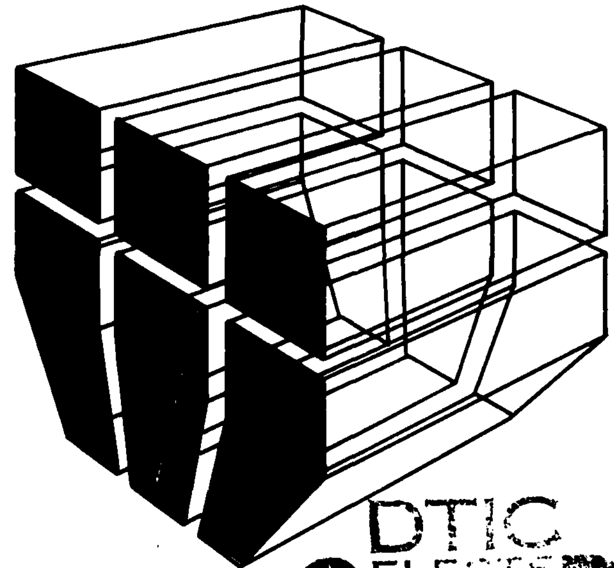
by
Navinchandra

The primary goal of this research effort was to develop a domain independent activity scheduling algorithm that would be able to handle *ad-hoc* constraints.

The activity scheduling problem is one of assigning tasks (activities) to objects (jobs) while adhering to time and resource constraints. Operations researchers originally had approached the problem using mathematical programming techniques. This approach, however, is poor at solving *real world* problems. *Real World* problems tend to be very large and are often too complex to represent numerically.

An algorithm is presented that is based on an *heuristic search* paradigm. It uses symbolic constraints to assist the search process. Functionally, the task is similar to that of linear programming. The scheduling problem is represented as a group of variables. Each variable has a corresponding set of possible values, called a *value set*. The aim is to assign each variable a value from its *value set* while adhering to the imposed constraints. The difference is that symbols rather than just numbers are dealt with. In so doing, the constraints are able to capture the nuances of complex domains.

A pattern directed constraint definition language called CDL-II is presented. The language is based on set theoretic operators and allows one to input constraints in an *ad hoc* fashion. The constraints are used to prune the search space through the mechanisms of constraint *generation, posting & propagation.*



DTIC
ELECTE
SEP 2 1986
S A

DTIC FILE COPY

Approved for public release; distribution unlimited.

86 9 02 111

The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official indorsement or approval of the use of such commercial products. The findings of this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN IT IS NO LONGER NEEDED
DO NOT RETURN IT TO THE ORIGINATOR

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CERL TM N-86/15	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) INTELLIGENT USE OF CONSTRAINTS FOR ACTIVITY SCHEDULING		5. TYPE OF REPORT & PERIOD COVERED Final
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Navinchandra		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS U.S. Army Construction Engr Research Laboratory P.O. Box 4005 Champaign, IL 61820-1305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS IAO 128-85
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE August 1986
		13. NUMBER OF PAGES 130
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Copies are available from the National Technical Information Service
Springfield, VA 22161

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

scheduling
activity scheduling
algorithms
ad hoc constraints

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The primary goal of this research effort was to develop a domain independent activity scheduling algorithm that would be able to handle *ad-hoc* constraints.

The activity scheduling problem is one of assigning tasks (activities) to objects (jobs) while adhering to time and resource constraints. Operations researchers originally had approached the problem using mathematical programming techniques. This approach, however, is poor at solving *real world* problems. *Real World* problems tend to be very large and are often too complex to represent numerically.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

BLOCK 20 (Cont'd)

An algorithm is presented that is based on an *heuristic search* paradigm. It uses symbolic constraints to assist the search process. Functionally, the task is similar to that of linear programming. The scheduling problem is represented as a group of variables. Each variable has a corresponding set of possible values, called a *value set*. The aim is to assign each variable a value from its *value set* while adhering to the imposed constraints. The difference is that symbols rather than just numbers are dealt with. In so doing, the constraints are able to capture the nuances of complex domains.

A pattern directed constrained definition language called CDL-II is presented. The language is based on set theoretic operators and allows one to input constraints in an *ad hoc* fashion. The constraints are used to prune the search space through the mechanisms of constraint *generation*, *posting & propagation*.

FOREWORD

This is a reprint of a thesis submitted to the Department of Civil Engineering, Massachusetts Institute of Technology, in partial fulfillment of the requirements for a Master of Science degree. The thesis supervisor was Professor David H. Marks.

This work was performed for the U.S. Army Training Support Center, Fort Eustis, VA, under Intra-Army Order 128-85, dated January 1985, for the Division Gunnery Model (DIGUM) Development project. The Technical Monitor is Maj. Robert Behncke, Army Development and Employment Agency, Fort Lewis, WA.

The research was supported by the Environmental Devision (EN), U.S. Army Construction Engineering Research Laboratory (USA-CERL). Dr. R.E. Riggins is Acting Chief of EN.

COL Paul J. Theuer is Commander and Director, USA-CERL, and Dr. L.R. Shaffer is Technical Director.

A1

Preface

Introduction

This thesis is an effort towards the development of a domain independent activity scheduling algorithm that can handle *ad-hoc* constraints. By combining the techniques of Artificial Intelligence and Operations research, a program has been developed that performs scheduling using a constraint assisted search approach. The program has a high-level constraint definition language which allows the user to input *ad-hoc* constraints.

Activity scheduling is the problem of assigning certain objects (jobs) to tasks (activities) while adhering to time and resource constraints. It is a popular area in Operations Research circles. Operations researchers, however, have been unable to solve large *real world* scheduling problems through mathematical programming techniques. Consequently, the whole area of heuristic scheduling has come to play an important role in this problem domain.

Several heuristic scheduling programs have been built. These programs tend to have the specifics of the associated domain *hard-wired* into the program. The only flexibility that these programs allow is the ability to change a few parameters in the constraint set. If a new, unanticipated constraint is encountered, the program's code needs to be changed. This is often not easily done.

To handle this problem, we present a constraint definition language called CDL-II. It allows the user to modify old constraints and add new ones at will. For example, let us consider the domain of job shop scheduling. The typical inputs to the system are the different jobs, the due dates, the quantities to be produced etc. Each job shop presumably has constraints which need to be adhered to. These constraints may range from due dates to machine preferences to problems regarding resource availability. Such constraints are not static and are ever changing. These new constraints may be of a totally unanticipated nature and would traditionally have been incorporated by painfully changing the original code. On the other hand, our framework allows one

to just write the constraint in CDL-II and enter it into the computer.

The algorithm

The techniques/technologies which have played an important role in the development of the ideas in this thesis are:

a) Branch and Bound Algorithms (from OR)

b) Planning Research (from AI)

c) Constraint Analysis (from AI)

d) Search paradigms and backtracking (from AI)

The algorithm presented in this thesis is based on the search paradigm. It uses constraints to prune the search space. Instead of following the *branch & bound* technique it has a *prune and then branch & bound* flavor.

This is done by using the constraints through the processes of *generation, posting and propagation*. The constraints are written in a constraint definition language called CDL-II. CDL-II is a production rule type, pattern directed language. This language is based on the set theoretic operators: intersection, difference, restriction and union. The other feature of CDL-II which has proved valuable is the ability to write constraints which, in turn, write constraints. This process of constraint generation is also pattern directed.

Currently the implementation uses chronological backtracking. Even though the scheduler uses contexts for the development of different branches, we have not used dependency directed backtracking [Sussman, 1978]. This is because our domain is not well suited for intelligent backtracking. This is in contrast to Sussman & Stallman's work wherein intelligent backtracking is facilitated by the scientific principles that govern the behavior of the objects in their domain, namely, electrical circuits.

Thesis Reader's Manual

The domain that has been chosen is that of Army Training Scheduling. The thesis starts off with a scenario (Chapter I) which describes the problem. The scenario, though anecdotal in nature, is representative of the problem and its complexity.

The second chapter introduces the problem and reviews the relevant literature.

The rest of the chapters discuss the use of constraints for activity scheduling. Two different implementations have been presented. The first implementation uses the constraints *passively* (chapter III). The constraints are used only for bounding the search. Such bounding is done only after the branching step is taken. Chapter III introduces a crude constraint definition language called CDL-I. It will be shown how CDL-I fails to capture the complexity of the problem and how the passive use of constraints is not a good approach.

NOTE: To understand the syntax of CDL-I & CDL-II the reader is advised to peruse the IMST manual (APPENDIX A). This manual describes the notion of an assertional database and explains the use of the IMST production rule language. CDL-II is completely written in IMST.

Having dispelled the concepts related to the passive use of constraints the thesis enters into a discussion about the active use of constraints. Chapter IV introduces the ideas behind this technique. It builds an intuitive feel for how constraints can be used for pruning the search.

Chapter V continues the discussion on the active use of constraints and goes on to describe the workings of the second implementation. This implementation is based on a constraint definition language called CDL-II. This language is entirely built in the IMST environment (Appendix A).

Appendix B presents a trace of the program run.

The author has assumed that the reader is familiar with planning research and constraint analysis. Here are a few papers that this thesis draws from. They make excellent background reading.....

1) Fikes R.E. (1970) "REF-ARF : A System for Solving Problems Stated as Procedures." Int.

J. of AI (1970) 1:27-120

2) Stefik M. (1981) "Planning with Constraints (MOLGEN: PartI)," Int. Journal of Artificial Intelligence, Vol 16, pp 111-140.

3) Sussman G.J., Steele G.L. (1980) "Constraints: A language for expressing almost hierarchical descriptions ", Int. Journal of Artificial Intelligence 14:1-39.

4) Fikes R.E. and N.J. Nilsson (1971) "STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving", Int. Journal of Artificial Intelligence, Vol 2 pp 189-208.

Contents

		Page
	DD FORM 1473	1
	FOREWORD	3
	PREFACE	4
I	**A Scenario**	10
II	**Introduction**	18
	II.1 Literature Review	
	II.2 Current methodologies	
	II.3 The Role of Search in Scheduling	
	II.4 Scheduling Army Training Activities	
III	**Passive use of constraints: - the first implementation**	32
	III.1 Introduction	
	III.2 Partial Schedules	
	III.3 The constraints	
	III.4 CDL-I: Performace Issues	
IV	**Active use of constraints: - some intuitive ideas**	40
	IV.1 Introduction	
	IV.2 Instantiation	
	IV.3 Generation & Posting	
	IV.4 Propagation	
	IV.5 A Classification for Constraints	
V	**Active use of Constraints: - The second implementation**	51
	V.0 Introduction	
	V.1 The representation	
	V.2 The Constraint Definition Language: CDL-II	
	V.3 The Algorithm	

V.4 Backtracking

VI Future Directions 72

VI.1 Exploiting the Flexibility of CDL-II
VI.2 Handling Multiple Heuristics in Scheduling
VI.3 Towards a system architecture

References & Bibliography 78

Appendix A IMST User's Manual 80

Appendix B Trace of a run (CDL-II) 102

Chapter I

A Scenario

This chapter sets the stage for the application domain used in this thesis. An anecdotal description of the domain is presented. It is representative of the complexity we had to deal with. We have also tried to give the reader a flavor of the kind of expertise current domain experts possess.

A

John Dale stood at the window watching trucks pass by. The air reeked of diesel and dust from the parched land. In his twelve years as range officer at Ft. Kami's Firing Range he had never seen such an influx of troops for fall training. Since the authorities had decided to move several battalions of the 5th Mechanized Infantry Division (MI) demand for firing ranges had been rising. The move was to be completed over the next few months, "things are going to get tougher" he said to himself.

In the distance, he could hear the sound of M-1 tanks firing at practice targets. The schedule on the wall told him that it was battalion 26 of the 5th MI Division on range_9. The convoy that had just passed the field office, 22 trucks of troops, were all headed for M-16 training on range_8. John Dale took great pride in his work and his acquired expertise in range scheduling and coordination. The boys who just passed him were headed for range_8, but he knew it would be safe even though tanks were training in the adjacent range. Everything had been worked out, the firing directions, the safety spans and the schedule conflicts.

John was concerned about the problems the new battalions might bring. He was going to be up against a very tough resource problem. The next master schedule was due in a month, and he anticipated problems with having to schedule so many more battalions.

The operations research group down at Ft. Kami had been helping John prepare the master

plan every quarter. He would send them a listing of all the battalions and the amount & type of training activities to be carried out by each battalion. A schedule would be returned in under 2 weeks.

The schedule was not always very useful. It did not take into consideration all those *real-life* problems that cropped up time and again. John remembered the time he spent with Mark Maser, a young eager systems analyst from Ft. Kami. He had told Mark all about the firing ranges and rules of thumb used in scheduling. He had told him about how certain activities can be carried out only on certain ranges and how the safety span considerations can change the way a schedule is built. Mark had been a good listener and sure enough, he came back in 3 months with a really impressive scheduling program. Since then whenever John had had some new requirements he'd just call up Mark and the changes would be made in a matter of days. Mark was gone now, had found a job in Silicon Valley. There was nobody to make changes anymore. The other programmers never seemed to have the time, nor did they seem to understand the nuances of the problem at hand.

B

The phone was ringing, John turned around to watch his assistant Fred Rufus pick up the phone. Fred reached for a pad and started making quick notes. It was that time of the quarter when the battalions filed their training request forms. A training request form containes information about the training activities the battalion wishes to perform, including preferences about timing and precedences. Many officers called in by phone to relay some exceptions and special requirements. Fred walked up to the scheduling board and stood there looking intently at. He scratched his head.

John Dale looked on. Fred was quickly becoming adept at his work, but not good enough to be allowed to make any changes without an endorsement from John. Fred better learn the ropes soon enough. It took John several years before he could get a handle on all the considerations required to run a firing range safely and smoothly. He knew the ranges, their characteristics; the

weapon systems and their characteristics like the back of his hand. His first and foremost concern was safety, never should two battalions be scheduled to work on the same firing range on the same day. Every time he scheduled a training activity he had to make sure that the safety spans of the current weapon system did not interfere with any other training activity scheduled on an adjoining range. He had all this information on his finger tips. Fred still had to refer to the range maps and safety span traces to schedule an activity.

John walked up to Fred, "What's the matter Fred? "

" Captain Roger Mason of battalion twenty-six, MI four called in and said that he would like his men to train with battalion 9 of MI three for Mortar firing next quarter."

"Two battalions together? That's something new"

" Battalion 26 of MI four is scheduled for Mortar on range 22 for the first week of December, but battalion 17 of the fifth MI are on range21 during that period. I cannot figure out how I can get 26 and 9 on adjoining ranges."

"Why?"

"For one thing, range23 is not suited for Mortar fire and range 21 is the only choice"

John cut in, "Can you move battalion 17 elsewhere?"

"No, they had requested completion of Dragon Qual before Christmas and I do not see any other free time windows."

It was a tough problem. John wished Mark was around. He could have just called Mark and told him about the change. He remembered the time when Central Training Command (CTCOM) had issued an order to perform cyclic training. CTCOM had found that certain training activities are most effective when carried out in a cyclic basis. That is, the total annual amount of training on certain activities was to be divided into 5 or 6 sessions. Each session was then to be scheduled such that no two sessions were less than one month apart nor were they more than 2 months apart.

It took Mark two weeks to make this change. John shuddered at the thought of what might happen if a directive as radical as the cyclic one were to be sent out by CTCOM today. John longed for a system that would be able to handle ad-hoc requirements.

C

John walked back to his desk to finish up a Memo he was preparing for CERL (the Corps of Engineer's Construction Engineering Research Lab.). Two researchers from CERL had come over last week to discuss the development of a Intelligent Scheduler which would be able to handle ad-hoc requirements. They had told him about how the scheduler would be able to handle changes and would store all the rules of thumb he was trying to teach Fred.

John was preparing a memo describing the scheduling problem in the form of a small representative example. He was to include a list of constraints.

The afternoon wore on, trucks were rolling past the range office. John Dale was smiling, he had just finished the Memo:

D

MEMORANDUM

From: John Dale, Range Officer, Ft. Kami

To: Bob James, CERL

Date: April 2, 1985

Here is a small representative example of the scheduling problem. I have included some constraints as you had requested.

Consider a firing range having three ranges:

range_A

range_B

and range_C

Refer figure I.1

There are three battalions:

bat_A

bat_B

bat_C

and there are three activities:

act_A

act_B

act_C

Let us assume a time period of two weeks:

days = { 1 2 3 4 5 6 7 8 9 10 11 12 }

Now we have several constraints on the problem:

Constraint: C1

A battalion can do only one activity in one time period (usually a day).

Constraint: C2

Conflicts: There shall be only one battalion on a particular range on a particular day.

Constraint: C3

If any battalion is scheduled for activity act_B then that battalion should not be scheduled for anything on the very next day. This is because act_B is very exahusting.

Constraint: C4

There are only certain ranges that can host certain activities:

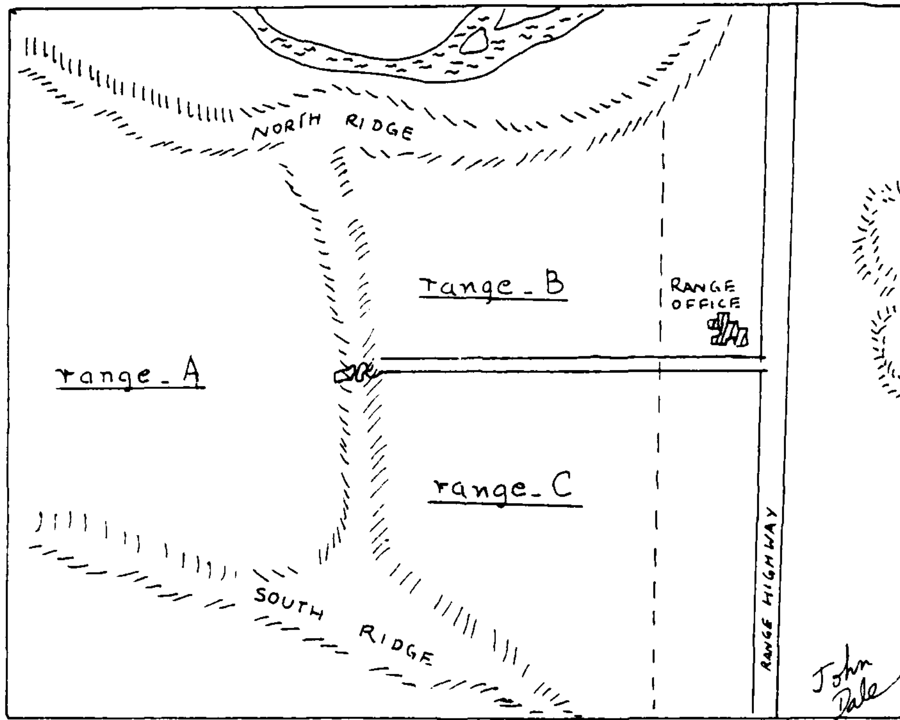
Constraint#	Activity	Legal Ranges
C4_one	act_A	range_A range_B
C4_two	act_B	range_C
C4_three	act_C	range_B

Constraint: C5

The battalions shall perform the activities according to the following frequences:

	activity		
	act_A	act_B	act_C
bat_A	3	0	0
bat_B	1	1	1
bat_C	1	2	0

Constraint: C6



FIRING RANGES

FIGURE I. 1

There is a cyclic constraint on activity act_A.
This means that a battalion should perform act_A at regular intervals. If the first schedule date is x then the next date should be after x+2 but before x+4.

Constraint: C7

Safety spans: When act_A is carried out on range_A, the safety spans requires that it is unsafe to schedule anything on range_C.

Chapter II

INTRODUCTION

II.0 Introduction

In this chapter we shall set the stage for the intelligent use of constraints in scheduling problems. We shall also review the relevant literature with includes references from both Artificial Intelligence and Operations Research.

Scheduling had originally been looked upon as a problem suited to mathematical programming techniques. This, however, is not true; the complexity and size of *real-world* scheduling problems has moved research towards more tractable algorithms involving heuristic search paradigms. Our hypothesis is that a domain independent scheduler can be built which would only use constraints to guide the search. Scheduling, per se, has been tackled by several researchers in AI, the most significant application is a system for Job-Shop scheduling called ISIS [Fox M.S., 1983].

We would like to remind the reader that scheduling has been researched by Management Scientists and Operation Researchers for several years. Some very promising results have emerged form such work. We now embark upon a survey of some of the work relevant to our interests.

II.1 Literature Review

Parts of this section (II.1) has been adapted from [Fox, M.S. 1983]:

Management Science

"Management science research in scheduling has focussed on understanding the variety of scheduling environments that exist, and constructing s heduling algorithms specific to them. Four types of "shops" are distinguished in the literature:

* single machine -single operation
* parallel machines - single operation
* flow shop series of machines - multiple operations
* job shop network of machines - multiple operations

A job is defined as having:

* one or more operations
* a processing time for each operation
* a due date

And the utility of scheduling is
measured in terms of:

* lateness
* flowtime
* tardiness
* makespan

It was recognized early in management science that scheduling is an example of a constraint satisfaction problem which could be optimally solved using mathematical programming techniques. Integer programming approaches, while theoretically valid are useless practically. One branch of research focuses on the attainment of optimal results, but algorithmic complexity has restricted these results to the one and two server cases. The achievement of these results requires the removal of much of the constraints, and the focus on single criterion for measuring schedule efficacy."

Artificial Intelligence

The area of Planning Research is the most relevant part of Artificial Intelligence to scheduling problems. The basic idea of using search to perform problem solving has been used both by AI researchers and by Operations Researchers. The use of Constraints in Planning research has proved very promising (Stefik Mark, 1981). Search is carried out within a space of possible solution states for a state that satisfies a set of pre-specified requirements. A state can be changed into another state by applying a heuristic operator to it. Planning can be viewed as a form of heuristic search. The first problem in creating a planning system is to generate the states relevant to reaching the goal.

"Given a description of the initial state, goal state, and a set of operators, the operators can be iteratively applied to the initial state, and its solution path of operations, or plan. Depending on the 'strength' of the operators, the space elaborated can be large or small; however the better heuristics generate smaller search spaces and find the solution faster. Planning, and related research, has focussed on a number of issues: for instance, choosing the state to elaborate next, choosing which operator to expand a state, and choosing alternative state representations and operators."

Robot planning is the most popular area in planning research. The STRIPS system (Fikes & Nilsson, 1971) represented operators with pre and post conditions.

This thesis has borrowed several ideas from a constraint analysis system called REF-ARF. (Fikes, 1970). Its task was similar to that of linear programming. Given a set of linear equations as constraints, it makes value assignments for all the variables. Instead of doing a brute force search for a set of bindings that satisfies all the constraints (equations), it used the constraints to reduce the generated binding set. Hence, the system can be viewed as a classical *generate and test*, where the system was able to take the constraints and use them in the generator to reduce the size of the search space. This thesis is an extension of REF-ARF wherein the constraints are symbolic in nature and are pattern directed.

After the work on STRIPS came a very important and interesting planning program called NOAH (Earl Sacerdoti 1975). By using hierarchical plan generation, Sacerdoti was able to implement an intelligent planner. Taking a cue from NOAH, Austin Tate (Tate A, 1977) developed NONLIN. It is a non-linear planner for generating Project Networks.

At about the same time some very interesting Constraint Analysis work was in progress at MIT. Stallman & Sussman (1978) developed the concept of dependency directed backtracking. A electrical circuit analyzer called EL was developed. EL used the concepts of constraint posting and propagation together with intelligent backup.

In 1981-82 Mark Stefik worked on an Intelligent Planner called MOLGEN (Stefik 1981). It is a planner for Genetic Experiments. MOLGEN uses constraints through the processes of generation, posting and propagation. This work was one of the most significant contributions to symbolic constraint propagation.

There are only a few AI scheduling systems:

> ".... One of the few AI scheduling systems was in the domain of train scheduling (Fukumori, 1980). It used a constraint-based approach to determine the arrival and departure times of trains. .... A second AI scheduling study was that of Vere (1981). In it plans are constructed, and times associated with each step in the plan. A sophisticated algorithm for time propagation based on interactions is described."

A third system is ISIS, a factory shop-floor scheduler. It is a constraint directed search

program which uses beam search to generate schedules. (Fox M.S. 1983)

Lastly there is a Intelligent Scheduling Assistant (ISA) project underway at the AI Technology Center at DEC. It is a rule based scheduler and is built in OPS-5. [Orciuch Ed, Frost John, 1985]

II.2 Current Methodologies

In this section we shall describe two existing approaches to the scheduling problem. The first one is a branch and bound method used for producing optimal schedules for a multi resource-constrained case. The second is a constraint directed search methodology used for Job Shop scheduling.

II.2.1 Branch & Bound [Stinson Joel, et.al. 1978]

The Stinson algorithm uses a branch and bound cycle to develop a tree of choice nodes. In order to keep the problem of tractable size, pruning is carried out using :

- Dominance Principles

- Lower bound pruning.

As the search tree is being expanded, dominance rules prune off nodes that are statically inferior to other generated nodes or are subsets of them. Once a set of candidate nodes are generated it is pruned using a lower bound *estimate to complete*. This estimate is calculated using heuristic projection techniques.

As long as the *estimate to complete* is a lower bound on the real completion time, the algorithm can be guaranteed to give optimal results.

The major problem with such an approach is that it cannot handle ad-hoc or complicated constraints. Our research efforts are toward a system which can handle any realistic constraint that is tossed at it.

II.2.2 Constraint Directed Search

[Fox, 1983]

The ISIS system, built at the Robotics Institute of the Carnegie-Mellon university, performs constraint directed heuristic search; constraints are used to bound and guide the search (scheduling) process.

We shall now describe the process of constraint directed search. The methodology described

here may look similar to that of ISIS. This discussion however, is not an attempt to explain ISIS, it is an attempt to familiarize the reader about the current techniques used.

Consider a scheduling problem wherein several jobs need to be scheduled in a machine shop. Each job has a set of activities that need to be carried out on it. These activities are essentially a sequence of operations to be carried out on several machines.

We start the scheduling by picking up the job with the highest priority and start scheduling it. While scheduling a job we start with its first operation, say 'op1'. It is possible that 'op1' may be carried out on one of several machines. To represent such choices a activity-network for each job is prepared. In figure II.2.2.1 we can see that job J1 has three operations : op1, op2 and op3. Operation op1 may be carried out on machine mc1 or mc2 and so on.. For operation op3 machine mc5 may be chosen in place of mc4 and mc6 taken together.

The search tree is then developed using this activity-network. Figure II.2.2.2 shows the development of such a search tree. At point 'A' we were dealing with operation 'op1' and have the choice of either taking mc1 or mc2. Going one step further we have four time periods to choose from for each machine. The schedule developed up until the choice of the time period is called a partial schedule. We now need some evaluation function by which the nodes can be weighted. This evaluation function, as the readers can see, is very critical to the efficacy of the search paradigm. These evaluations are heuristic in nature and much research has gone into the study of these evaluation functions.

Setting aside this issue for a the moment we proceed to develop our search tree. Assume we decide to do operation op1 on machine mc1 in time period 't1'. This places us at node 'H' with a choice between nodes: I,J,K or L. The hatched line in the figure shows the path taken till node K, where time period 't4' was chosen for 'mc3'. This can be shown as a Gantt chart: figure II.2.2.3

After having scheduled Job J1 fully, the system chooses the next job and performs the same process. If there ever is a clash at a machine, the system tries to backup and choose another

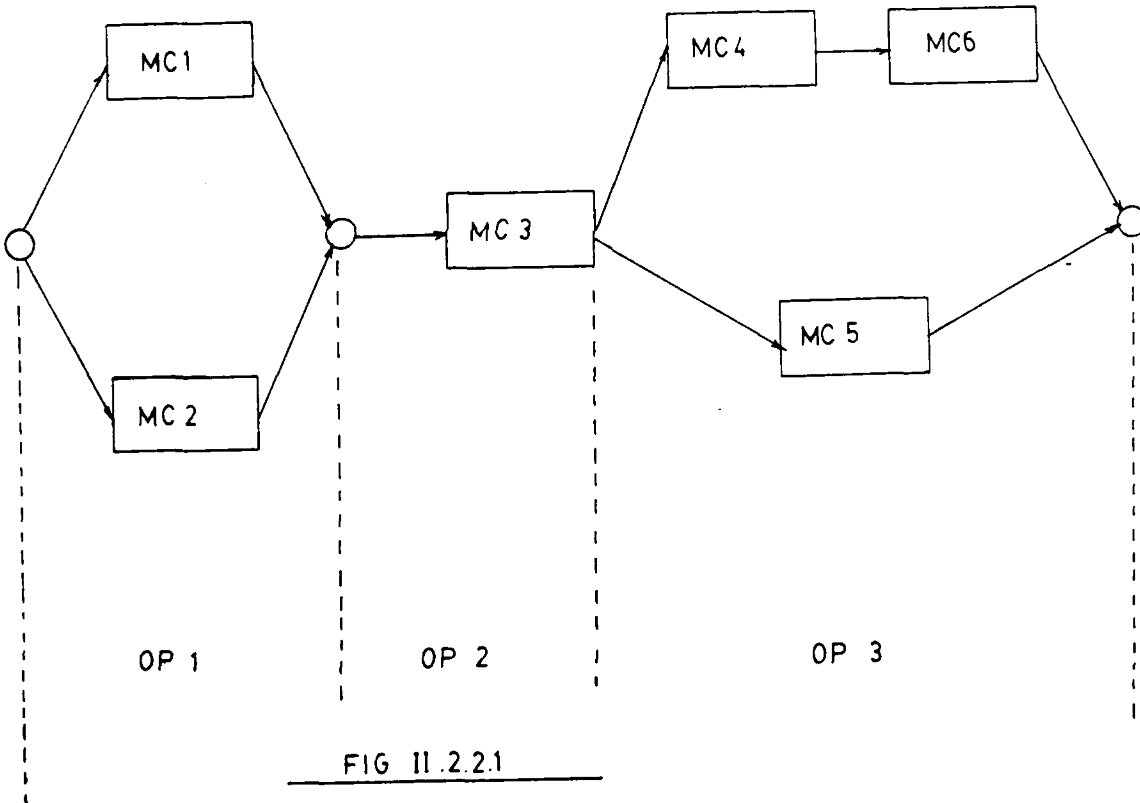


FIG II.2.2.1

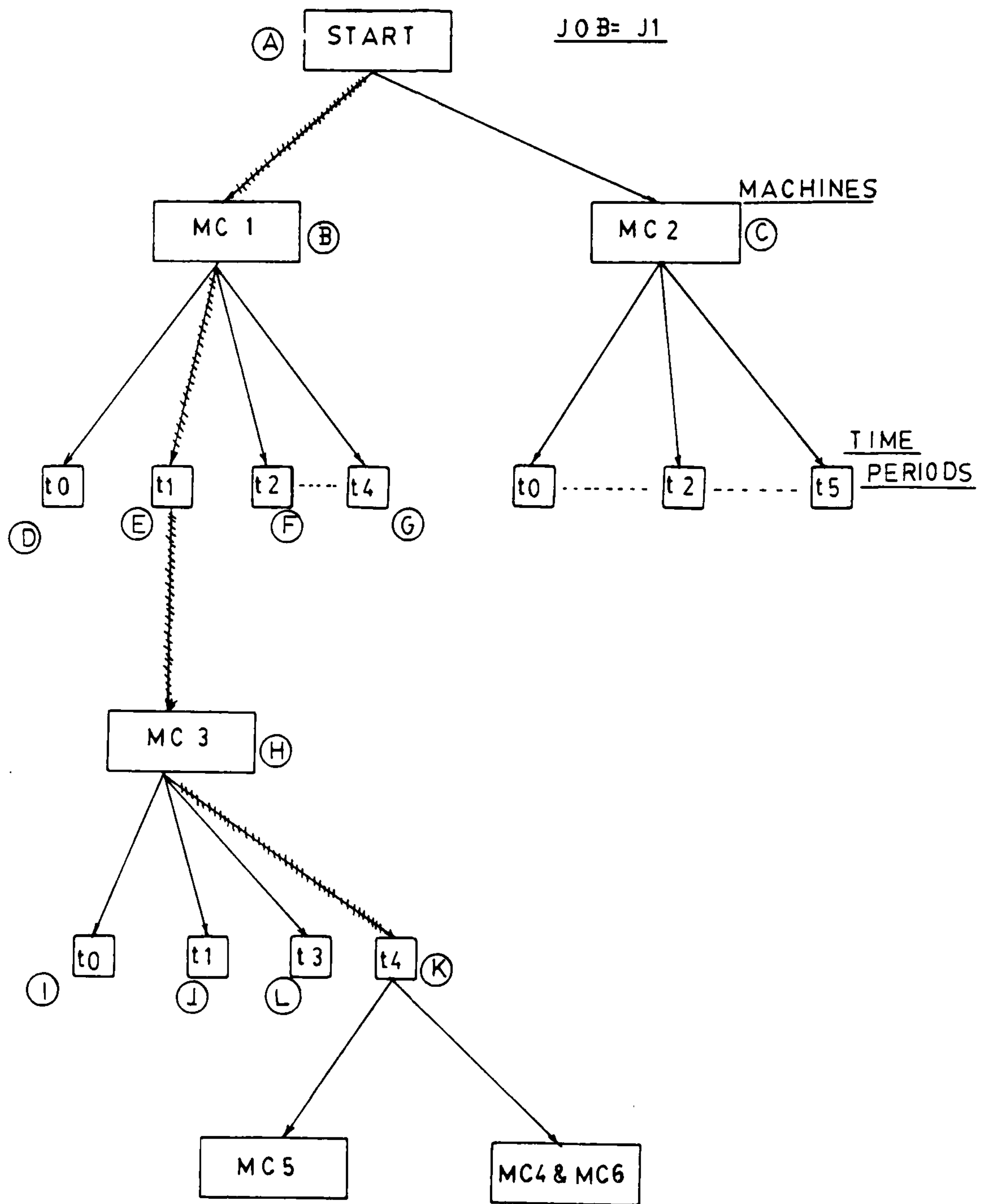
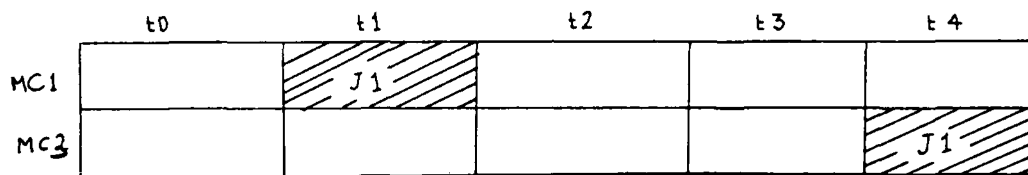


FIG II.2.2.2



PARTIAL SCHEDULE AT NODE K IN FIGURE II.2.2.2

FIGURE II.2.2.3

machine. If this is not possible, the job with higher priority gets scheduled and the other one goes back to the bag of unscheduled jobs.

That was the basic idea of scheduling using a search tree. We now discuss the method of choosing a node when there are several viable choices. A evaluation function is used to find the best choice.

In a constrained problem one can evaluate the nodes on the basis of the extent to which each node satisfies the constraints. If each constraint has an utility then the total utility at a node i is given by:

$$\text{Utility} = \sum_{i} (\text{Utility of constraint } i) * (\text{level of satisfaction of } i) \quad \ldots.[1]$$

Such an utility can be calculated for each node, that is, for each partial schedule represented at that node. The above evaluation function does not make an effort to "look forward". In other words, the evaluation is based upon the choices made up until the current node 'n'. Let us call this utility g(n). If the choice of a node is said to be based on a value called f(n), then the evaluator just described is:

$$f(n) = g(n) \quad \ldots [2]$$

The branch and bound technique's f(n) is different

$$f(n) = g(n) + h(n) \quad \ldots [3]$$

where h(n) is a heuristic estimate of the total *cost to complete.* If the h(n) is guaranteed to be a lower bound on the actual cost to complete then the search is guaranteed to terminate at a optimal solution. [Nillson 1971]

The subsequent sections touch upon the problem we are trying to address in our scheduling project. The domain is that of scheduling the training activities of troops at an army installation.

II.3 The role of Search in Scheduling

We start this section with an explanation of the scheduling problem in a domain independent fashion. A typical scheduler can be thought of as a system which answers the following questions

1 Who

2 Where

3 What

4 When

In the job shop scheduling domain the 'who' is the job, 'where' is the machine, 'what' is the operation and 'when' signifies the scheduled time. [Note: More dimensions (e.g. 'which' for resources) can be added to suit the domain in question]

Figure II.3.1 shows how this kind of a scheme might work. Each cycle of choices "who -> where -> what -> when ..." is called a *scheduling cycle.*

The search tree, as shown in figure II.3.1, is bound to become very large. To avoid a combinatorial explosion we could restructure the representation as in figure II.4.1 Changes in search or control of constraints are all carried out by heuristic rules. This formalism allows us to change the performance of the system by changing the heuristic rules.

II.4 Scheduling Army Training Activities

The research presented in this paper is all the product of the development of a scheduling system for army training. Translating the framework presented in section III.1, we have:

who = battalion

where = range

what = training activity

when = date

In addition to this basic framework, we have a large set of constraints that need to be satisfied. The constraints specify the conditions to be met in the final solution. It is possible that

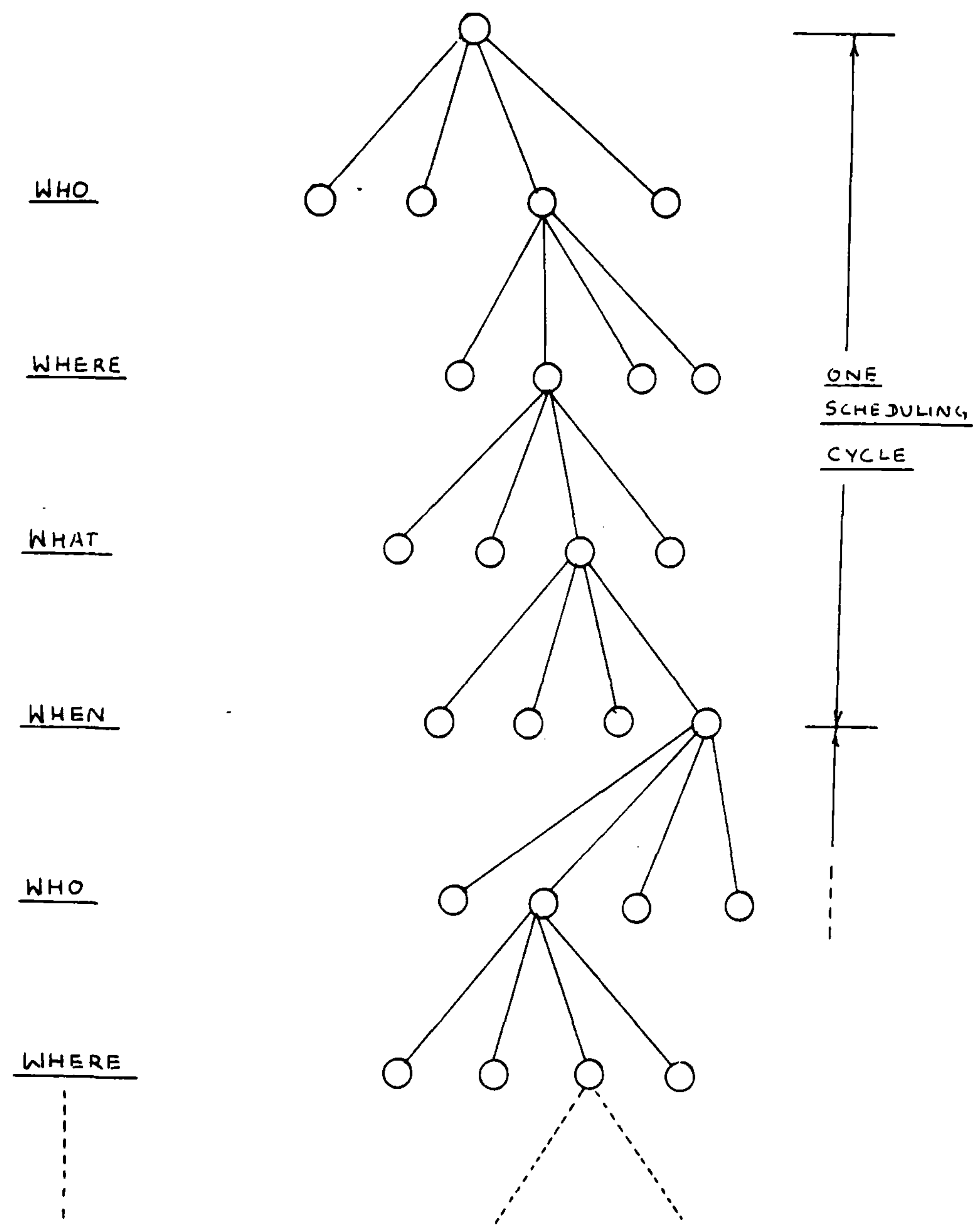


FIGURE II.3.1

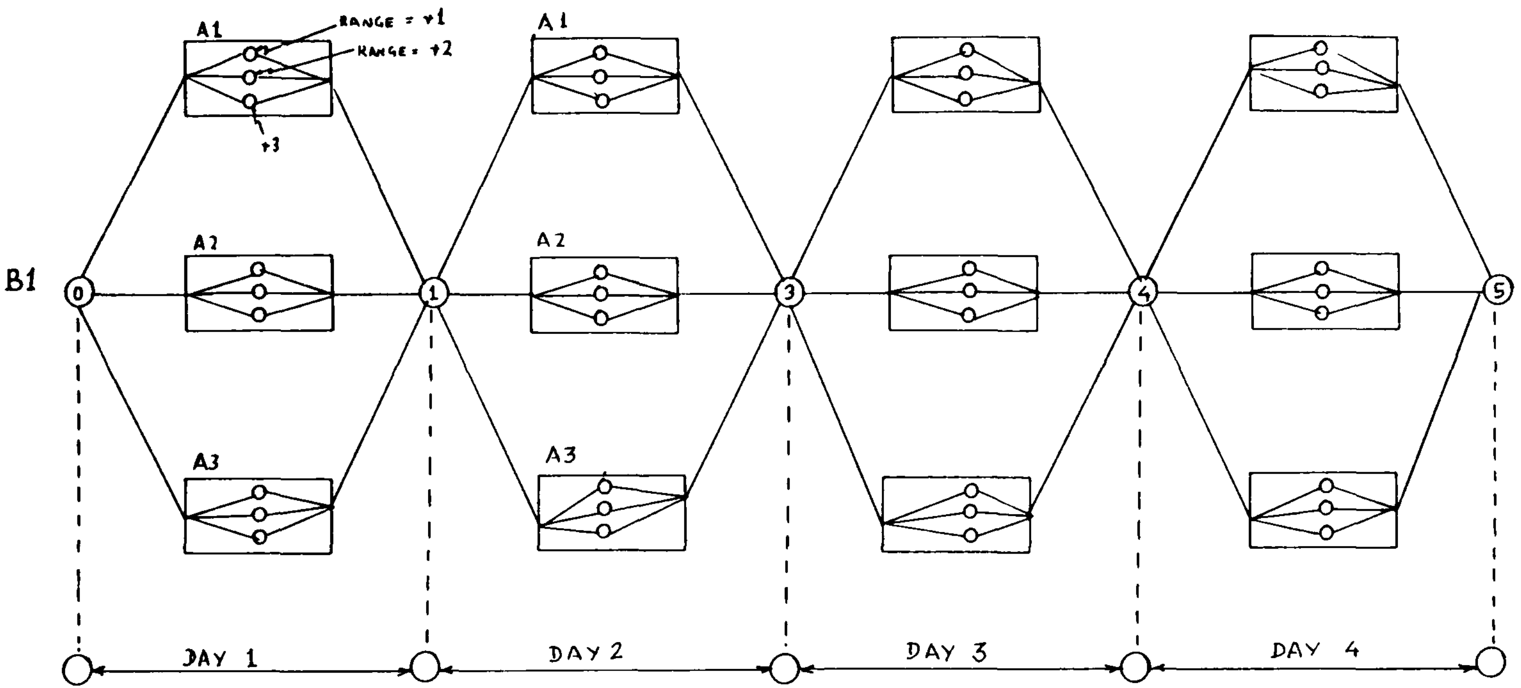


FIGURE II.4.1 ACTIVITY NETWORK

there are several solutions to the problem. Our first objective now, is to find any of the satisficing solutions.

For this domain, a activity network is drawn for each battalion. The network is shown in figure II.4.1 The figure shows the activity network of a battalions called 'B1' . The activities to be chosen from are : a1,a2 and a3. The firing range is to be chosen from among r1, r2 and r3. The search tree has been reduced in size by assuming sequential scheduling. The activity network has choice nodes at the beginning of each day.

Once an activity has been chosen for a particular day, the next step shall be the choice of the range where the activity should be carried out. The ranges available to each activity is dictated by the nature of the activity and the size of the range. (e.g. If a range is small, it cannot support weapon systems which require large safety spans.)

Having presented the basics of schedule generation using the Search paradigm, we conclude this introductory chapter. The following chapters start the discussion about the role of constraints in scheduling.

Chapter III

Passive Use of Constraints:

The First Implementation

III.1 Introduction

It is our aim to develop a scheduling system that will be able to handle ad-hoc constraints. This chapter presents some of the representations used for schedule generation and discusses the passive use of constraints.

Constraints are said to be passive when they do not play an active role in guiding the search process. The constraints are used only after a partial schedule is generated. In essence it is a case of the *Generate and Test* paradigm.

III.2 Partial Schedules

As stated in section II.3, the scheduling problem can be solved by search techniques. The product of each scheduling cycle is an assignment of a battalion to a range for a particular training activity on a particular day. This is represented as a predicate called *scheduled*. The form of the predicate is thus:

(scheduled {battalion_name} {activity_name} {range_no} {time_period})

Each such assignment is called a schedule element. The complete schedule consists of several such schedule elements. Consider a scheduling cycle of the following configuration:

battalions: {bat_A bat_B bat_C}

activities: { act_A act_B act_C }

ranges: { range_A range_B range_C }

time periods: { 1 2 3 4 5 6 7 8 9 10 11 12}

A search tree is shown in figure: III.2.1 . The hatched lines in the figure shows us that battalion: bat_A is going to perform activity: act_A on range_C on day: 8. In addition it shows

that battalion: bat_B will perform act_C on range_B on day: 6.

The scheduling elements developed in the figure are:

(scheduled bat_A act_A range_C 8)

(scheduled bat_B act_C range_B 6)

By going deeper in the search tree the scheduling elements start representing a more detailed schedule. The constraints are used to check the schedules prepared by the search process.

A depth first search technique was employed. The first implementation was built with the idea of using constraints in an *after the fact* fashion. For successful completion, the schedule produced by the system had to satisfy all the constraints. Every time the path is expanded, it is checked. There are three outcomes of such a check. The check function returns a message to the scheduler telling it the status of the schedule generated.

1 all_satisfied

This means that all the constraints are fully satisfied, and that the schedule is complete.

2 continue

This message is passed back when the current schedule is found to be partial. When the schedule is partial, all the constraints will not have been invoked. However, those invoked will be satisfied. (Note: a constraint is invoked when it's premises are true.)

3 failure

. As soon as a violation occurs, failure is indicated. The failure message causes the scheduler to backtrack to the last scheduling decision. This action is called chronological backtracking and is generally very inefficient.

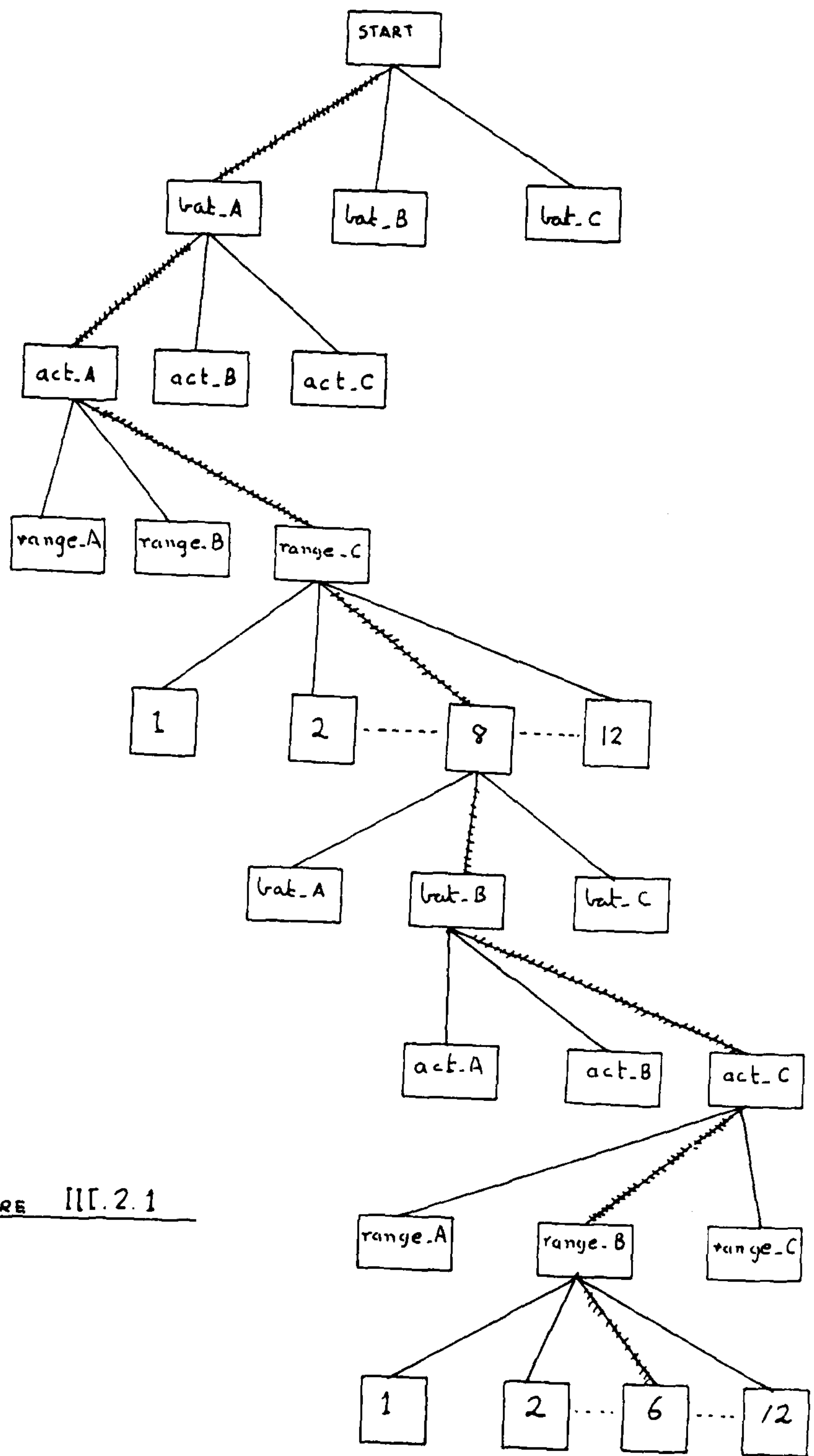


FIGURE III.2.1

The procedure

1.0 Form a stack of constraints

2.0 Unstack the first constraint and test it against the current partial schedule.
Add the returned message to a *results* list.

3.0 If any one of the results is *failure* then fail and start backtracking .

4.0 If not end of constraints stack, go to 2.0

5.0 If any one of the results is: *not-applicable* then continue the search.

6.0 If all results are 'satisfied' then return the message: 'all_satisfied' and terminate the search.

III.3 The constraints

The reader would have noticed the use of message called 'not-applicable'. This signifies the situation where a partial schedule cannot satisfy all the constraints. Let us now examine what 'not-applicable' means:-

Each constraint has three parts:

(1) Name of the constraint

(2) The applicability pattern/patterns

(3) The tests

When a constraint is to be checked against a schedule, its applicability is checked first. If it is applicable, then the tests are executed. The constraint is deemed satisfied if all the tests return 'true'.

The constraints are written in a crude (first cut) constraint definition language called CDL-I. The CDL used in a later implementation is more powerful than the one presented in this section. Before we go further, let me set the stage for CDL-I and present a few examples.

CDL-I makes use of a pattern matching and variable binding facility. It's patterns act directly on the schedule elements. Extending the example problem presented in scenario (Chapter I) we shall set up some constraints for the problem.

The syntax of a typical constraint is:

```
(constraint     (name  -name- )
                ( -pattern- )
                ( -tests- )      )
```

Let us now translate some of the constraints into CDL-I.

The constraint C1: "A battalion can do only one activity in one time period".

```
(constraint     (name C1)
                (>bat  >act  >range  >day)

--> (equal (sigma (<bat >act >range <day)) 1.0))
```

The above constraint has the name 'C1' and has a pattern:

```
(>bat   >act   >range   >day)
```

and a test (for testing the generated schedule).

The program has a stack of such constraints. It picks up a constraint and tests it. The first step is to match the pattern against a database. The use of the symbol '>' (do not view it as a greater-than sign) means that the attached word is a variable. This variable can be bound to any value in the process of pattern matching. For example, if we match the pattern:

```
(bat_A >x1 >y1 >z1)
```

with the schedule element (stored in the data base):

```
(bat_A act_A range_A 8)
```

then the matcher will match bat_A to bat_A , x1 to act_A, y1 to range_A and so on.

Consequently the bindings will be thus:

x1 = act_A

y1 = range_A

z1 = 8

Once bound, the variables can be used in the testing part of the constraint.

Now for an extended example: let us assume that there is a partial schedule that looks like this:-

```
( (bat_A  act_A  range_B  8)
  (bat_B  act_B  range_C  4)
  (bat_C  act_C  range_A  6)
  (bat_B  act_A  range_A  4) )
```

There is a problem in this schedule. Battalion: bat_B has been scheduled to do two different things on the same day (i.e. day: 4).

The constraint C1 has to catch this. It has in its test a statement that says : "for each battalion, the day assigned to each schedule element has to be unique to that schedule element". In other words, the total number of schedule elements in the database that have the same battalion name **AND** the same day should be unity.

For example, let us choose bat_B. We start at the top with the first schedule element for bat_B:

```
(bat_B act_B range_B 4)
```

The constraint C1 will have the following bindings: act = 'act_B', range = 'range_C' and day = 4. It now knows that for bat_B there should not be any other schedule element with day = 4. This is regardless of the activity and the range. In some sense, the total number of schedule elements that match:

```
(bat_B >act >range 4)
```

should be unity. The summations is carried out by the sigma function in the CDL-I constraint shown above. The pattern has two variables >act and >range in it. For this reason it will match any element which has the string "bat_B" in the first place, anything in the second and third places and "4" in the fourth place. We can see that the above pattern will match the partial schedule twice. Consequently the test will fail.

Let us look at the constraint C1's code again. It had a test part with the pattern:

```
(<bat >act >range <day)
```

in it. This pattern has two variables >act and >range. It has a new symbol "<" . This symbol means that the value of the variable should be instantiated. In other words the the above pattern gets converted to

```
(bat_B >act >range 4).
```

This is the basic idea behind CDL-I's pattern matcher. We have used this method of binding and instantiation because we later will write constraints which are able to write constraints.

The constraint C2: "There shall be only one battalion on a particular range in a particular time period."

```
(constraint   (name  C2)
              (>bb >aa >rr >dd)

--> (equal (sigma (>b  >a <rr <dd) )
           1.0))
```

Constraint C4: The ranges which are suited to each activity:

```
(constraint (name C4_one)
            (>bat act_A >range >day)
        --> (or (equal <range range_A)
                (equal <range range_B)) )

(constraint (name C4_two)
            (>bat act_B >range >day)
        --> (equal <range range_C))
```

CDL-I is not good for some complex constraints. A CDL-II has been developed which allows for the intelligent use of constraints.

III.4 CDL-I: Performance Issues

The performance of the CDL-I based search mechanism was *very* poor. The program took several hours to run. Slightly bigger problems caused serious problems and was full of backups. It always seemed to be looking down the wrong path! It has been estimated that if this strategy is used for a full fledged problem, it may require several months of CPU time before it can terminate successfully. So much for the passive use of constraints.

The purpose of this chapter is to give the reader a feel for the motivations for moving towards the intelligent use of constraints. The development of such intelligent techniques is the essence of this thesis.

Chapter IV

Active Use of Constraints:

- Some Intuitive Ideas

IV.1 Introduction

This chapter adopts a highly intuitive approach to the concept of *active constraint utilization*. In Chapter III we saw how the *generate & test* method of search fails to deliver. In this chapter we show how constraints can be used for search pruning. We present a framework wherein constraints are not *hard-wired* into the scheduler but are input via a constraint definition language (CDL). Consequently, the constraints can be changed by the user as and when adjustments are required. Such flexibility is essential to real-world scheduling systems.

The constraints and the data are the only domain dependent parts of the program. The constraints are essential to the operation of the scheduler. If there are no Constraints at all, the program will do nothing. We need to tell it that it is a scheduler!

Consider the constraint shown below:-

[Note: The CDL used here is in plain English only for the sake of brevity, a more formal development is presented in Chapter V]

```
[Constraint:   Battalion #42 shall perform:
          [1]   300 hrs of tank_training
          [2]   50 hrs of Dragon_qual
          [3]   50 hrs of M16_sustain
          [4]   1 CALFX
          [5]   ....
]
```

Given the above constraint, the system may start scheduling all the activities on the same day and maybe on the same range. More constraints need to be added to avoid this problem:

```
[Constraint: There shall be only one activity per
          battalion per day]
```

[Constraint: There shall be only one battalion per
range per day]

These two constraints will avoid clashes and ensure safety.

Having given the reader a flavor of what we mean by constraints; we now embark upon a series of examples to illustrate the operations on the constraints. The basic operations are:

[1] Instantiation

[2] Generation

[3] Posting

[4] Propagation

[5] Relaxation (not examined in this paper)

[6] Satisfaction

IV.2 Instantiation

A constraint is said to be *instantiated* when any of the variables within the CDL are bound to objects in the domain. If there is a constraint which says that every battalion shall qualify on the M-16:-

[Constraint: For all values of 'x' ,where 'x' is a
battalion, that 'x' shall perform 50
hours of M-16 qualification]

The variable 'x' is then free to be bound to any battalion in the database. Once this generic constraint is bound, it is *posted* against the corresponding object.

IV.3 Generation & Posting

Constraint posting can be initiated either by a pre-specified global constraint or by a generated one.

(1) A pre-specified global constraint is one that is always
true (at least they are intended to be so).

Here's an example:-

[Constraint: There shall be no training on Sunday]

(2) A conditional constraint is generated in the process of solution:

[Constraint: IF tank-training is scheduled for range16

THEN It is not safe to train on the adjoining ranges: range15 and range17.]

Such a constraint remains passive until its pre-conditions are met.

Preference constraints are also posted against objects in the database. One may prefer to do a FTX type activity during the non-winter months or one may prefer not to schedule M16_qual on range12. These constraints are posted at the appropriate location on the activity network.

IV.4 Propagation

The first significant use of constraint posting and propagation in planning was done by Stallman & Sussman (1977). Later, Mark Stefik of Stanford built an excellent system called MOLGEN (Stefik 1981). His work has shown us how symbolic constraints, through the process of posting and propagation can help in complex domains.

There are several types of constraint propagation relevant to our work.

a) Forward Propagation

After a constraint is posted, it may be propagated. Consider a constraint which requires us to perform an activity 'act6' for 3 successive days:

[Constraint: IF 'act6'is scheduled on a day 'x'
THEN 'act6' shall be scheduled on days x + 1
and x + 2 also]

Figure IV.4.1 shows us how such a change might occur. As soon as 'act6' was scheduled on day11, the above constraint was activated and it was propagated to day12 & day13. The next scheduling cycle will start on day14.

b) Cross Propagation

In section IV.1 we introduced a constraint which required that only one battalion can be on a range at a time. Figure IV.4.2 shows us how such a constraint is propagated from one job to

another. If the Armor Battalion10 is scheduled on range 'R1' on a particular day, the constraint is propagated to other battalions and the range is deactivated.

Another kind of cross propagation is diagonal in nature. Consider the following preference constraint:

[Constraint: IF tank_training was scheduled on range
'x' on day 'y'

THEN It is preferred to reschedule
tank_training on range 'x' on
day ('y' + 1).]

This constraint captures the fact that the setup costs of targets on a range for a particular activity should be translated to the next activity scheduled on that range. Figure IV.4.3 shows how this may be done in a two battalion situation. As soon as tank_training was chosen for Bat1 at range R3 on DAY116 it propagates 'diagonally' across to the other battalion and increases the *preference level* of R3 for tank-training. In other words, if tank_training is chosen for Bat2 on day117, then range R3 would be preferred.

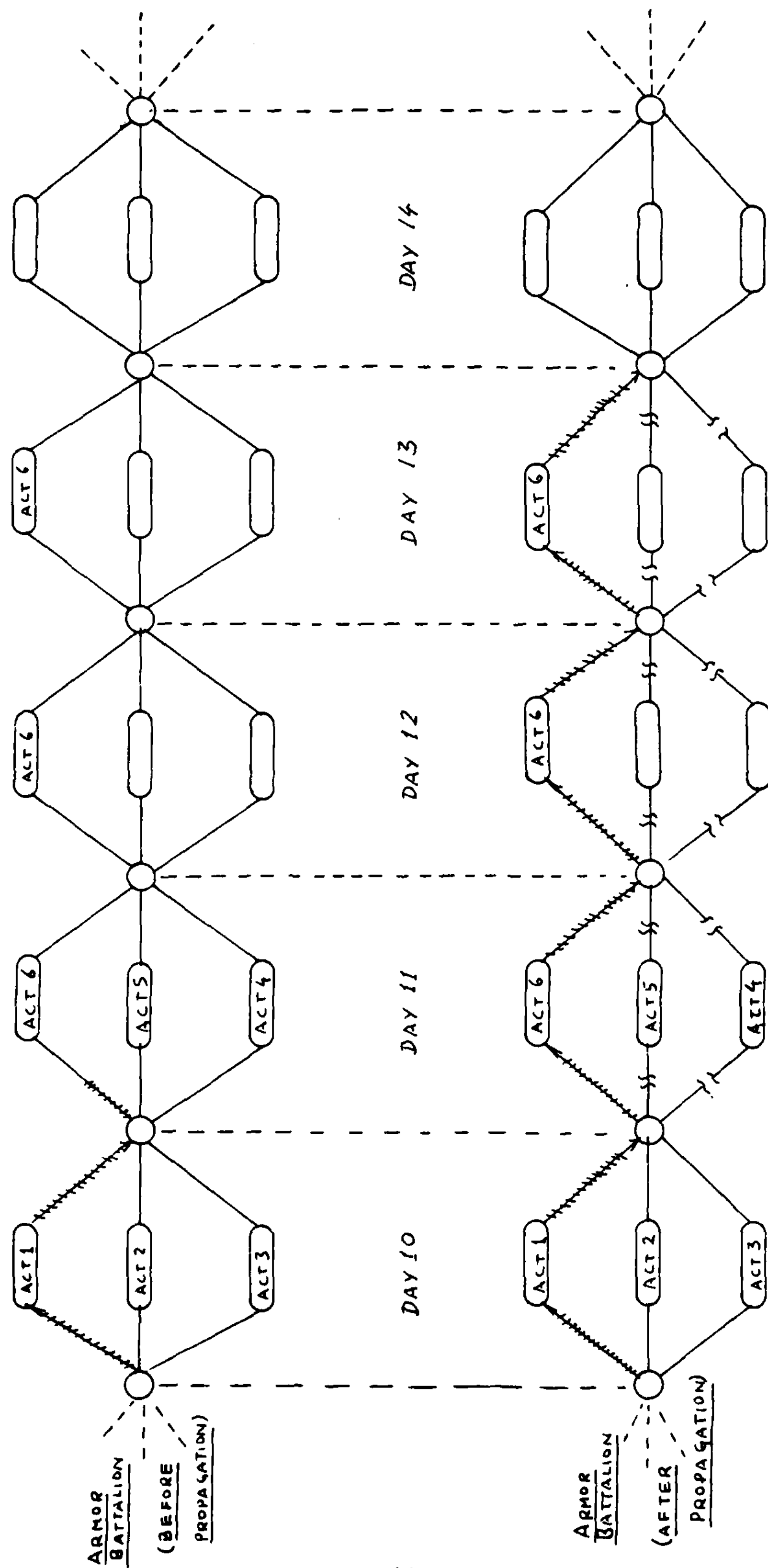


FIGURE IV.4.1

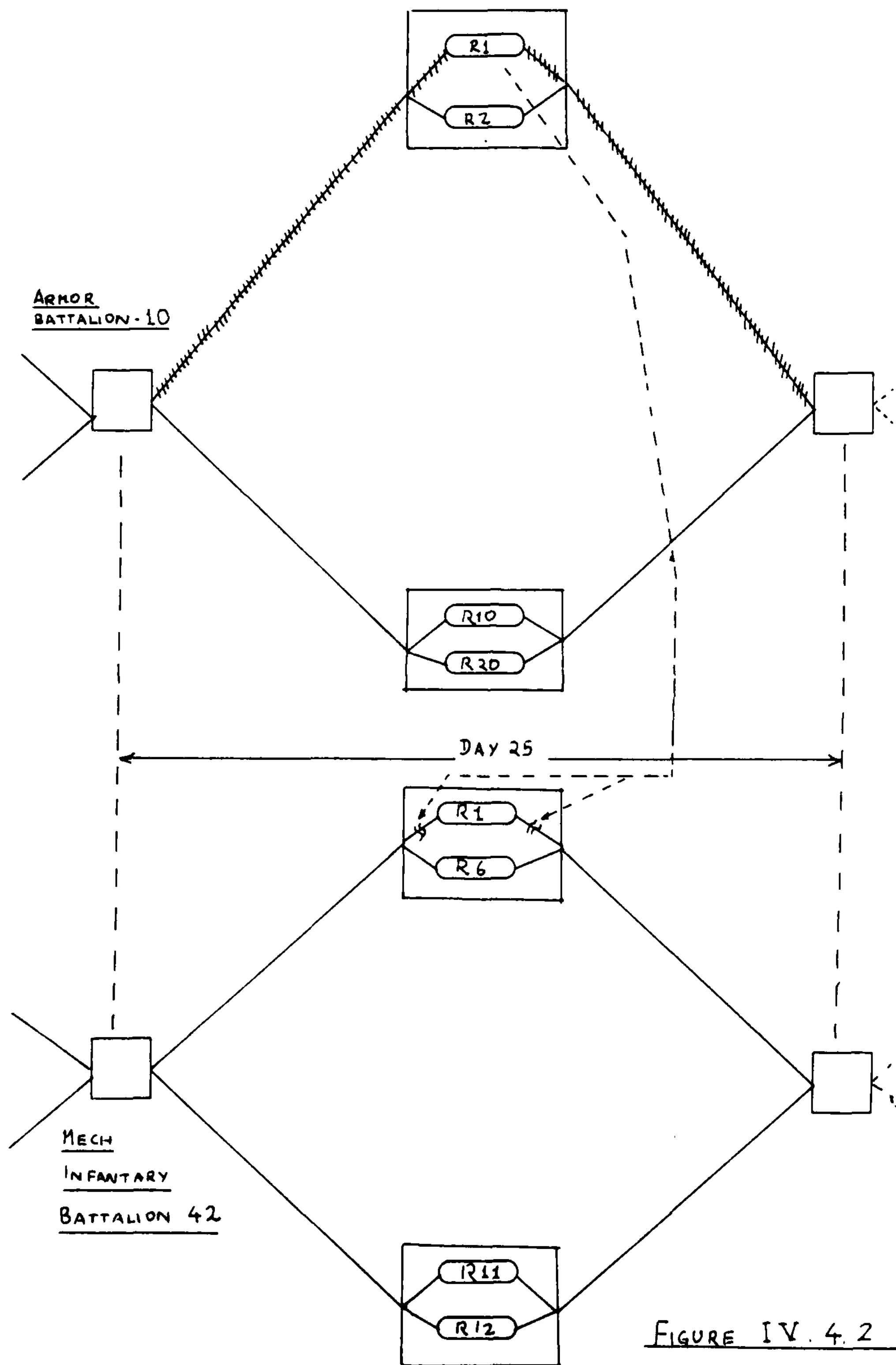


FIGURE IV. 4. 2

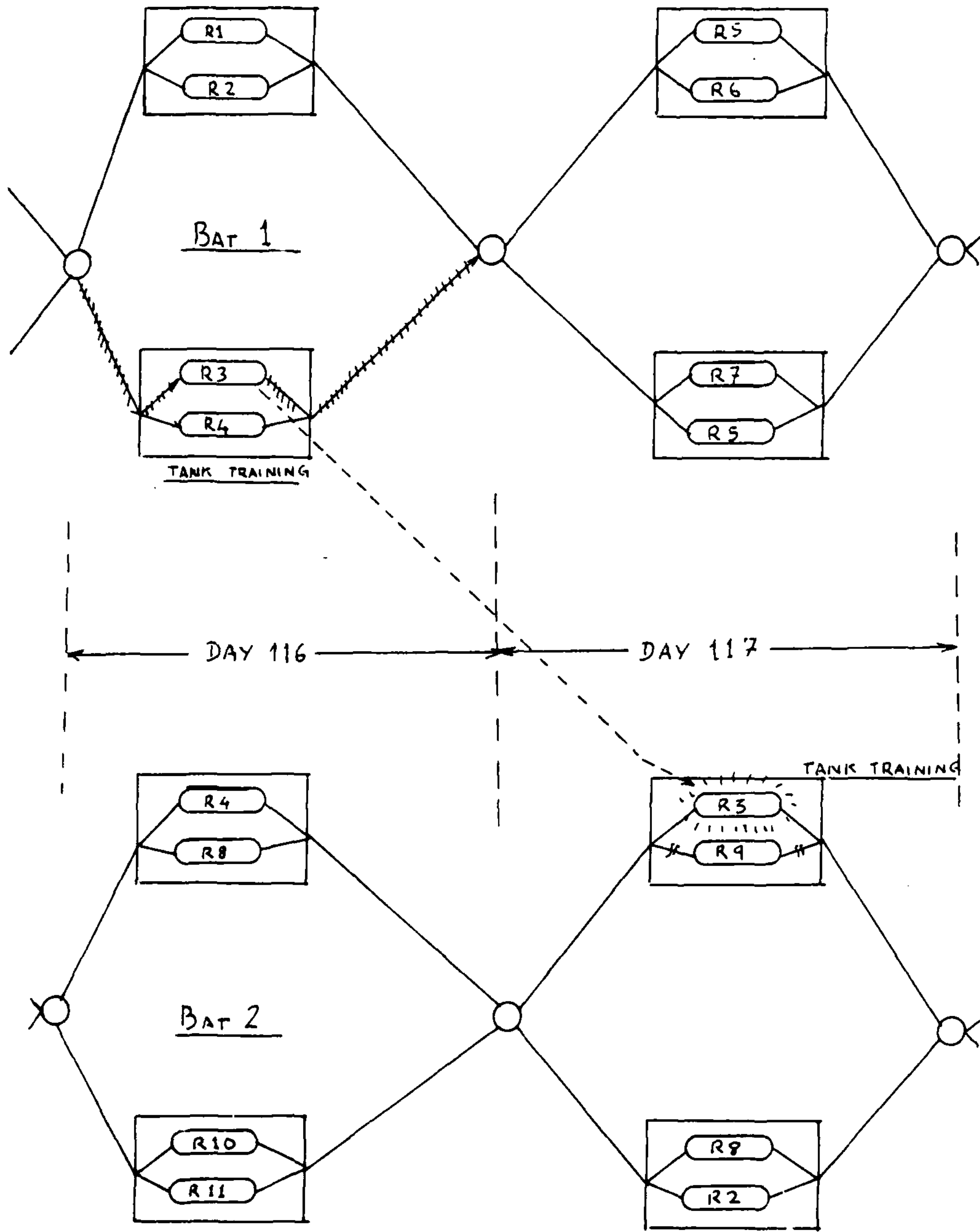


FIGURE IV. 4. 3

c) Backward Propagation

We shall examine two forms of backward propagation, the first type is used for due-date constraints.

[Constraint:Duedate1 : Armor Battalion10 should complete
all training by Oct 15]

Backward propagation (i.e. backwards from the due-date) is to be used only if there was a failure in the forward propagation. This mode of propagation is similar to forward propagation, but in reverse.

The second type of backward propagation is very interesting. In the domain of troops training, one has to design well balanced training programs. It is required that certain proficiency levels are maintained among the troops in different weapon systems. In keeping with this, a cyclical constraint is imposed. Such a constraint requires that certain training activities should be carried out at regular intervals, throughout the year. This ensures that the troops neither train on one weapon system all at once, nor have long time gaps between training sessions of a particular type. Figure IV.4.4 [Source: Eilts, Wright, Houck 1984] shows a plot of proficiency levels vs time. In order to maintain a steady level of proficiency in any weapon system, troops should be trained on that system periodically.

[Constraint: Cyclical : IF a battalion is scheduled for
tank_training on some day 'd'

THEN it should not be rescheduled for
tank-training over the next 30 days
after 'd'
AND it should be rescheduled in less
than 60 days after 'd']

Assume that a typical tank-training session was 5 days long and there were to be five such sessions in a year. Figure IV.4.5 shows this constraint as a bunch of blocks and springs. The blocks signify the training sessions and the springs allow us to incorporate some slack. The length of a block represents the time period of the corresponding session and the length of the spring between two blocks is the time gap between sessions. The level of compression of the springs is a measure of

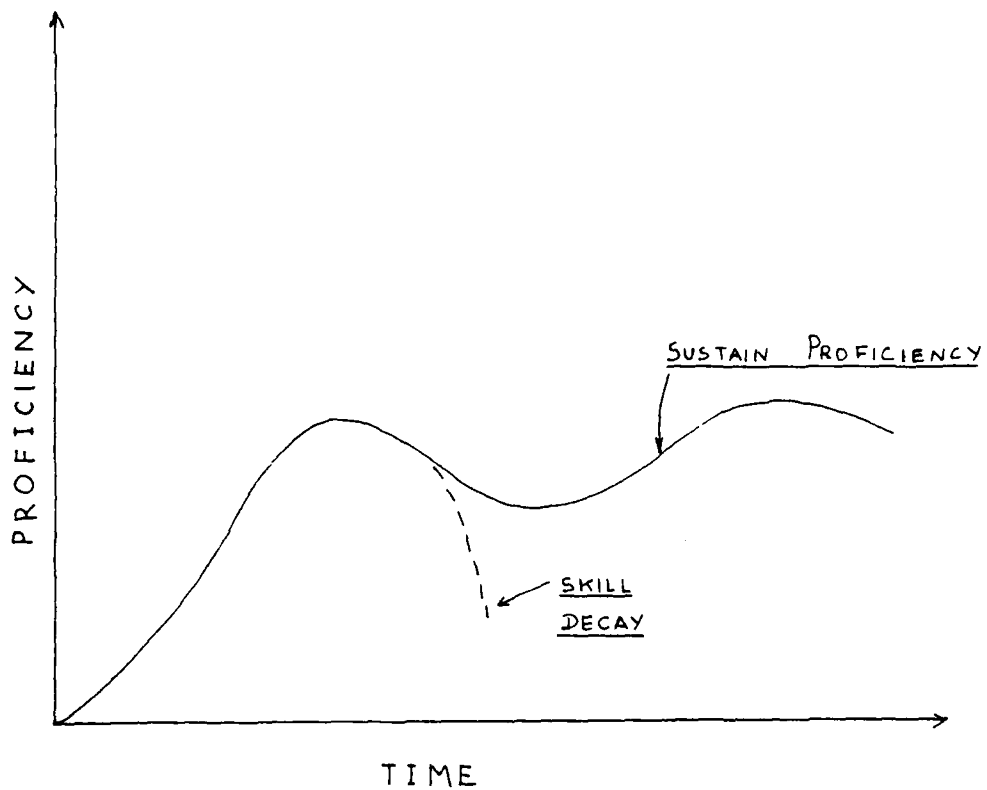


FIGURE IV.4.4 LEARNING AND FORGETTING CURVES FOR TRAINING (SOURCE: EILTS, WRIGHT & HOUCK 1984)

deviation from the constraint's tenets.

Once the scheduling starts, a constraint like the one in figure IV.4.5 *lurks* about the vicinity of the planning networks and waits to be invoked. This constraint will come into effect only after the first training session is actually scheduled. As the scheduler advances along the planning network the cyclical constraint's block&spring system gets pushed back. As the end of year approaches the block&spring system gets pushed up the 'utility mountain' (figure IV.4.6). As the scheduling front moves forward without scheduling tank-training it gets tougher and tougher to push back the block&spring system. This is corrected by a backwardly propagated constraint. As shown in the figure, constraint 'A', propagates backwards and increases the preference level of tank-training for the next day.

IV.5 A classification for constraints.

Constraints can be classified on the basis of the effects that they have on the schedule elements and their variables. A constraint that uses data of several schedule elements is called multi-element in nature. The multiplicity of variables is the other dimension.

	Single Element	Multi Element
Single variable	SV/SE	SV/ME
Multi variable	MV/SE	MV/ME

The most complex constraint is the MV/ME (Multi-variable/multi-element) type. It involves several scheduling elements and places constraints on a more than one variable. By using constraint generation, it is possible to convert complex constraints into simpler cases. The simplest case is the SV/SE case. The methods used to convert a constraint from one class to another is presented in the next chapter.

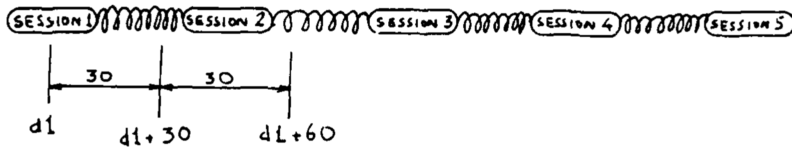


FIGURE IV. 4.5 BLOCK & SPRING SYSTEM

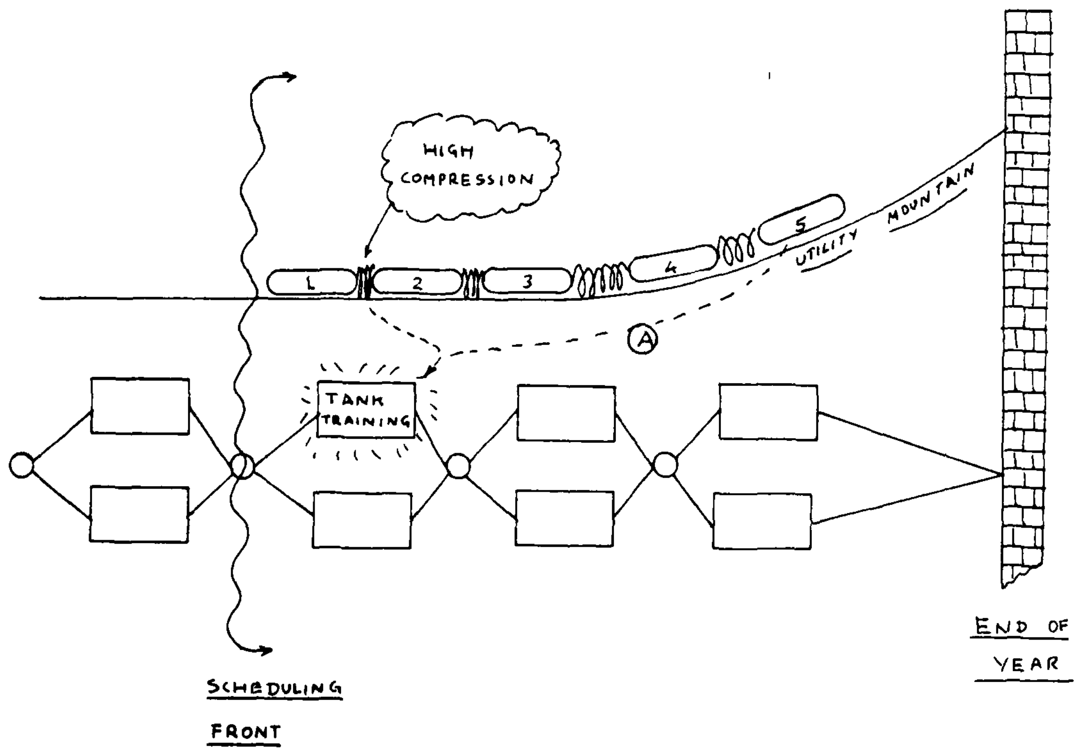


FIGURE IV. 4. 6 BACKWARD PROPAGATION

Chapter V

Active Use of Constraints

- The Second Implementation

V.0 Introduction

Having given the reader a intuitive feel of how constraints may be used for scheduling. We now proceed to understanding the workings of the second implementation. The second implementation uses constraints through the powerful mechanisms of *constraint generation, posting & propagation*. Unlike the passive case, this technique tends to be lot more efficient. Large problems could be solved in reasonable (few hours) time.

A new and improved constraint definition language called CDL-II was used. CDL-II is built wholly in a production system (similar to YAPS) called IMST. (Refer Appendix A)

V.1 The representation

The basic task performed by the scheduling algorithm presented in this thesis is similar to that of linear programming.

The problem is set up as a large group of variables. Each variable has a corresponding *value set*. These *value sets* are lists of atoms. Through a search process one of the atoms is chosen from the *value set*. This is done for each variable. When all the variables are assigned to *unary* (single valued) sets, the schedule is deemed to be complete.

In Chapter IV we mentioned that a schedule consists of several *schedule elements*. In the program the schedule elements have variables in them. For example, the *schedule element* (bat_A act_C range_A 9) says that battalion bat_A will perform activity act_C on range_A on day: 9. This schedule element will now be represented as:

```
(bat_A act1 range1 day1)
```

where act1, range1 and day1 are variables that are attached to battalion: bat_A. The

corresponding variable bindings are:

act1 = act_A

range1 = range_A

day1 = 9

Each schedule element represents one schedule instance. That is, it represents a unique set of variable assignments. These scheduling elements are stored in a data base. The element can be put into the data base by using the **assert** function.

For a better understanding of the concepts of assertional databases and of pattern directed inferencing, the reader is urged to read Chapters 1 & 2 of Appendix A

V.2 The constraint Definition language CDL-II

CDL-II is based on the concepts of fundamental set theory. Each schedule element has variables in it. For example a schedule element:

```
(bat_A act0019 range0019 day0019)
```

has three variables act0019, range0019 and day0019. These variables have associated sets of values from which one value has to be chosen.

The constraints are used to trim and focus these value sets and thus help in pruning the search space. The physical action of the constraints on these sets are set theoretic in nature:

- Intersection
- Subtraction
- Union
- Restriction

These basic operations allow us to manipulate value sets in various ways. The first three operators are obvious. Restrict is used to filter a *value set* based on a particular predicates or some

testing function. For example, if one wants to restrict a list of the first 10 integers to only those that are greater than 6:-

```
(restrict       '(1 2 3 4 5 6 7 8 9 10)
                '(gt $$$ 6)
)
```

The $$$ symbol signifies the list which has to undergo restriction. The result of the above restriction will be (7 8 9 10).

The other functions available in CDL-II are:

(set_value -var- -list-)
this sets the value of a variable to a particular list.

(get_value -var-)
retrieves the latest value of a variable.

(selected? -var-)
checks to see if all the variables in the supplied list are selected variables. A selected variable is one which has a value set with only one value in it.

With these functions it is possible to write constraints. Once again, picking up the example problem from part **D** of chapter I.

(a) **Constraint: C1**

In English:

"A battalion can do only one activity in one time period ."

In CDL-I:

```
(constraint C1 (>bat  >act  >range  >day)

        test (selected? <day)

--> (constraint   C1_aux   (<bat  >a1  >r1  >d1)

        test (not (equal (quote <day)  d1))

        --> (set_value  d1
                (subtract  (get_value  d1)
                           (get_value (quote  <day)))
                )))
```

Here is a good example of constraint generation. The constraint C1 produces a secondary constraint called C1_aux. Once generated, the C1_aux performs SV/SE (refer Section IV.5) propagation. Note that C1 is SV/ME constraint and it gets converted into the SV/SE case.

Let us complement this with an example. If we have a partial schedule:

```
1:  (assert bat_A  act001  range001  day001)

2:  (assert bat_A  act002  range002  day002)

3:  (assert bat_A  act003  range003  day003)

4:  (set_value 'act001 '(a1) )

5:  (set_value  'act002  '(a1) )

6:  (set_value  'act003  '(a1) )
```

Figure: V.2.1

This way of setting up the data explicitly lists down the number of scheduling elements required. Notice that the scheduling elements are added to the database via assertions which consist of a battalion name followed by three variables. The database produced will be:

```
( bat_A  act001  range001  day001)

( bat_A  act002  range002  day002)

( bat_A  act003  range003  day003)
```

The corresponding variable bindings:

Variable	Binding
act001	(a1)
act002	(a1)
act003	(a1)

When the constraint C1 is executed it will generate the following new constraint.

```
(constraint   C1_aux  (bat_A  >a1  >r1  >d1)

          test        (not (equal 'day001  d1))


-->  (set_value d1

         (subtract (get_value  d1)
                   (get_value 'day001))) )
```

The other two constraints will have day002 & day003 in place of day001 above. C1_aux can match any element of battalion 'bat_A' except the one with day001 in it. It will then proceed to subtract the value from the current element's day set (bound to d1, above). For example if day001 was set to (8), then it would be deemed *selected* and will make constraint C1 generate C1_aux. In turn C1_aux will subtract out the value (8) from the value sets of all other day elments of that battalion.

(b) Constraint: C2

In English:

There shall be only one battalion on a particular range in a particular time period.

In CDL-II:

```
(constraint C2 (>bat >act >range >day)
  test (selected? <range <day)

--> (constraint  C2_one (>b2 >a2 >r2 >d2)
        test (not (equal a2 (quote <act)))
                 ; make sure its not the same one!

              (equal (get_value r2) (quote  <(get_value range)))

              --> (set_value d2
                    (subtract (get_value d2)
                              (quote  <(get_value day)))))

    ; the second constraint generated by C2 :-

    (constraint C2_two (>b3 >a3 >r3 >d3)
         test (not (equal a3 (quote <act)))
              (equal (get_value d3)
                     (quote  <(get_value day)))

              --> (set_value r3
                    (subtract (get_value r3)
                              (quote  <(get_value range))))) )
```

(The use of '!' is equivalent of the use of the comma for the backquote macro.)

Constraint C2 looks for any schedule element that has both the range & the day selected. Once this is done it generates two new constraints C2_one and C2_two. For example, if range: range_B is reserved for day 5, then C2_one will go around looking for any battalion that is scheduled for range: range_B, it will then remove day: 5 from the value set of that schedule element. C2_two does just the opposite.

Constraint C2 was multi-variable/multi-element whereas C2_one and C2_two are SV/ME .

Extending the data set in figure V.2.1 we have:

```
7:  (assert bat_B  act004  range004   day004)
8:  (set_value 'act004 '(act_A))
9:  (set_value 'range004  '(range_A))
10: (set_value 'day004   '(5))

11: (assert bat_B  act005   range005  day005)
12: (set_value 'day005  '(1 2 3 4 5 6 7 8 9 10 11 12) )
13: (set_value 'range005 '(range_A))
```

Figure: V.2.2

The statements 7,9 & 10 will cause constraint C2 to generate C2_one and C2_two, as the range & day are both selected for battalion bat_B, the new constraints that are generated are:

```
(constraint  C2_one  (>b2  >a2  >r2  >d2)

        test (not (equal  a2   'act004))
             (equal  (get_value  r2  '(range_A)))

-->  (set_value  d2
                (subtract (get_value d2) '(5))
        ))


(constraint C2_two (>b3  >a3  >r3  >d3)

        test (not (equal a3 'act004))
             (equal (get_value d3) '(5))

--> (set_value  r3  (subtract  (get_value r3)
                                '(range_one)))
)
```

Let us go through the process by which C2_two is tested against the partial schedule presented in figures V.2.1 and V.2.2. When C2_two comes to expression 11. The bindings will be :

b2 = bat_A

a2 = act005

r2 = range005

d2 = day005

The first test is: `(not (equal 'act005 'act004))`, the second one is: `(equal (get_value 'range005) '(range_A))`. Both tests are true. The final part expands to:

```
(set_value 'day005
        (subtract   '(1 2 3 4 5 6 7 8 9 10)  '(5)) )
```

The value of day005 is now set to a new value set without the value (5). The search proceeds with this new restricted value set.

(c)Constraint C3

In English:

If any battalion is scheduled for activity act_B then it should not be scheduled for anything in the immediately next period.

In CDL-II:

```
(constraint C3  (>bat >act >ran >dd)

        test (equal (get_value <act) '(act_B))
             (selected? <dd)


--> (constraint  C3_aux  (<bat  >act1  >ran1 >ddd)

        test (not (equal ran1) (quote <ran))

        --> (set_value  ddd
                (subtract (get_value ddd)
                        (list (+ 1 (car (get_value (quote <dd))))
                ))))
```

(d) **Constraint C4**

In English:

Activity act_A can be carried out on ranges range_A and range_B only

In CDL-II:

```
(constraint  C4  (>b1  >a1  >r1  >d1)
        test (equal (get_value <a1)  '(activity_A))
        --> (set_value <r1  '(range_A  range_B)))
```

Note that C4 is a SV/SE constraint and is directly applicable.

(e) **Constraint C5**

These constraints are expressed directly as assertions and set_value calls. Figures V.2.1 & V.2.2 show us how this may be done. This is probably one of the limitations of CDL-II. It requires the user to specify the actual number of schedule elements. Each assertion produces one schedule element with three variables, the constraints could be used to develop the elements but it would require some kludgery.

(f) **Constraint: C6**

The cyclic constraint produces a constraint network which allows for backward propagation.

In English:

There is a cyclic constraint on the activity called act_A. This means that a battalion should perform act_A at regular intervals. If the first scheduled date is x, then the next date should be after x+2 but before x+4.

The constraint is clearly MV/ME in nature. It should allow for both backwards and forward propagation, as discussed in section IV.4 . Being a MV/ME case we will have to reduce it to a SV/ME case by using the constraint generation technique we have been using till now. To avoid

excessive kludgery it was decided to present only the SV/ME case (after generation is done).

The constraint C6 is to be exposed on act_A, there are three occurrences of act_A (figure V.2.1). From the figure we see that the three variables day001, day002 & day003 are to be governed by the cyclic constraint. Assuming that there is a function called create_cycle which produces the following constraint, we have:

```
(constraint c6_one (bat_A act001 range001 day001)
                   ; being specific to battalion_one
                   ; could be done automatically

    -->

 (set_value 'day003
      (restrict (get_value 'day003)
               '(and (ge $$$ (+ (get_min_value 'day002) 2))
                     (le $$$ (+ (get_max_value 'day002) 4)))))

 (set_value 'day002
      (restrict (get_value 'day002)
               '(and (ge $$$ (+ (get_min_value 'day001) 2))
                     (le $$$ (+ (get_max_value 'day001) 4))
                     (le $$$ (- (get_max_value 'day003) 2))
                     (ge $$$ (- (get_min_value 'day003) 4)))))

 (set_value 'day001
      (restrict (get_value 'day001)
               '(and (le $$$ (- (get_max_value 'day002) 2))
                     (ge $$$ (- (get_min_value 'day002) 4)))))
               )
```

The functions get_max_value and get_min_value get the maximum and minimum value of a specified variable. The form of the constraint sets up a constraint network (Sussman G.J., G.L. Steele 1980). This network deals with value sets and has to maintain consistent propagation. As the values are numeric, propagation is simple and definitive.

Let us take up an example at this point. Extending the example in figure V.2.1 we have:

```
14: (set  'total_period '(1 2 3 4 5 6 7 8 9 10 11 12))
15: (set 'earlier  '(1 2 3 4 5))
16: (set 'later '(8 9 10 11 12))

17: (set_value 'day001 later)
18: (set_value 'day002 total_period)
19: (set_value 'day003 total_period)
```

Let us now walk through the propagation. The initial value sets are as below:

```
day001:        (8 9 10 11)
day002:        (1 2 3 4 5 6 7 8 9 10 11 12)
day003:        (1 2 3 4 5 6 7 8 9 10 11 12)
```

When C6_one executes, the first set_value expands to:

```
(set_value 'day003

           (restrict  '(1 2 3 4 5 6 7 8 9 10 11 12)
                      '(and (ge $$$ (+ 1 2))
                            (le $$$ (+ 12 4)) )))
```

The restriction causes the value set to shrink:

```
day003:        (3 4 5 6 7 8 9 10 1 12)
```

When the second part expands, we get:

```
(set_value 'day002  (restrict '(1 2 3 4 5 6 7 8 9 10 11 12)

                               '(and (ge $$$ 10)
                                     (le $$$ 16)
                                     (le $$$ 10)
                                     (ge $$$ (- 3 4)))
                          ))
```

```
        This will return:

        day002:        (10)
```

The next times C6_one is called we get:

```
        day001:        (8)
        day003:        (12)
```

This is really how constraint propagation alone could be used to come up with answers. Things do not always work out this way. Generally constraints reduce the value-sets to some smaller sets which then require search techniques. This is the topic of the next section.

V.3 The Algorithm

The underlying algorithm is that of search. The process of search consists of propagation-search-propagation cycles which terminates when all the variables have reached *selected* status.

The Algorithm:

1 Load data

2 Load Constraints

3 Carry out all the propagation possible.

Unless the propagation returns an error , continue propagation

IF a failure is reached

THEN backup and retract all the generated constraints.

IF the backup returns failure,

THEN announce schedule failure and stop.

4 After all propagation, choose the most constrained variable,i.e. the variable with the smallest value-set other than unary.

IF no such variable exists, THEN announce success and stop.

5 Expand (branch further down the tree)

6 Go to step 3

Figure V.3.1

There are two ways that the algorithm terminates:

(1) success

Success is reached when all the variables have unary *value-sets*. This condition is detected in step4 in figure: V.3.1 . A function called **get_most_constrainted_variable** searches from among the variables which have yet to be assigned. It returns that variable which has the smallest *value set* other than unary. This is done to reduce the branching factor.

2) failure

A total *failure* occurs when the *backup* (step 3) reaches the bottom of the scheduling stack. Currently the backup is chronological in nature. This is because non-chronological or dependency directed backup is not easily determined (Stallman, R & G.J. Sussman 1977). These ideas will be developed later in this chapter.

V.3.2 Contexts and their Tracking

Every time the scheduler passes through the *propagate-branch cycle*, it produces a new context. The new *values sets* that are assigned to a variable in each propagation step are all context dependent. Further, in order ot help backtracking the program stores a history of the *value sets* for each variable. In this way, the retraction of a decision is done by just undoing all the effects of the corresponding context. Contexts are represented by the symbol `cycle`. The progression of Contexts is represented by numerically increasing the cycle number: `cycle1 cycle2 cycle3`......

We will now go through an example which shows how new contexts are created and how they are used. Before we go further, lets look at the functions **set_value** and **get_value** once again and see what they *really* do. The value of a variable is actually stored as a push down stack. The **set_value** pushes the new value onto the stack along with the value of the current context. The **get_value** looks up the top of the stack and hence returns the latest value.

Consider a new example:

```
1:      (assert bat_A  act1 range1 day1)
2:      (set_value 'act1   '(act_A))
3:      (set_value 'day1 '(1 2 3 4 5 6))
4:      (set_value 'range1   '(range_A   range_B  range_C))

5:      (assert bat_A  act2  range2 day2)
6:      (set_value 'act2 '(act_A))
7:      (set_value 'range2 '(range_A))
8:      (set_value 'day2 '(5))

9:      (assert bat_B act3  range3  day3)
10:     (set_value 'act3 '(act_C))
11:     (set_value 'range3  '(range_A  range_B  range_C ))
12:     (set_value 'day3        '(1 2 3 4 5 6))
```

Figure: V.3.2

The above data assignments will translate into the variable assignments as shown in figure V.3.3 . Assuming that the current cycle number is cycle0, we have:

Variable	Value
act1	((cycle0 (act_A)))
day1	((cycle0 (1 2 3 4 5 6)))
range1	((cycle0 (range_A range_B range_C)))
act2	((cycle0 (act_A)))
day2	((cycle0 (5)))
range2	((cycle0 (range_A)))
act3	((cycle0 (act_C)))
range3	((cycle0 (range_A range_B range_C)))
day3	((cycle0 (1 2 3 4 5 6)))

Figure: V.3.3

Note how the cycle number is stored along with the *value-sets*. We are now ready to start applying the constraints. Using the same constraints as in chapter I (except constraint: C7) we get the the following new values. These values are put in on the top of the stacks of the corresponding variables:

Variable	New value added to the stack	Constraint name
range1	(cycle0 (range_A range_B))	C4_one
range3	(cycle0 (range_B))	C4_three
day1	(cycle0 (1 2 3 4 6))	C1

Note that during propagation the cycle number does not change. In the current

implementation the constraint numbers are not stored along with the propagated lists. If this practice were adopted, it could help in performing dependency directed backtracking.

Once all the propagation is complete, the program looks for the most constrained variable. This variable, by definition, is one whose latest *value-set* has the minimum number of values. The minimum number is however has to be greater than unity.

The variable 'range1' is the most constrained. The branches that are produced are: range_A and range_B. These two values constitute two different choices and are hence attached to new contexts. This is performed in two steps.

```
(set_value 'range1  '(cycle1  (range_B)) )

(set_value 'range1  '(cycle2 (range_A)) )
```

The stack for range1 now looks like this:

```
variable: range1

        (cycle2 (range_A))
        (cycle1 (range_B))
        (cycle0 (range_A range_B))
        (cycle0 (range_A range_B range_C))
```

Figure: V.3.4

The program attempts propagation once again. The latest context being cycle2. No effective propagation occurs. NOTE that we had dropped constraint C7 from the current constraint set. We have dropped C7 only momentarily and will reintroduce it later.

Once again, the most constrained variable is chosen, this time it is day1 with the value set (1 2 3 4 6). The stack for day1 looks like this:

```
variable: day1

                (cycle7 (1))
                (cycle6 (2))
                (cycle5 (3))
                (cycle4 (4))
                (cycle3 (6))
                (cycle0 (1 2 3 4 6))
                (cycle0 (1 2 3 4 5 6))
```

Figure: V.3.5

After the branching shown above, propagation is attempted. Once again, propagation does not occur. We enter the next branching stage by choosing day3.

```
variable: day3

                (cycle13 (1))
                (cycle12 (2))
                (cycle11 (3))
                (cycle10 (4))
                (cycle9  (5))
                (cycle8  (6))
                (cycle0 (1 2 3 4 5 6))
```

Figure: V.3.5

Once again no effective propagation occurs. We now look at a listing of all the variables and their values. The value of a variable, as returned by the function **get_value** is it's latest value (top of the stack) regardless of the associated cycle number.

```
Variable                  (get_value  -var-)
name
---------------------------------------------

act1                      (cycle0 (act_A))
day1                      (cycle7 (1))
range1                    (cycle2 (range_A))

act2                      (cycle0 (act_A))
day2                      (cycle0 (5))
range2                    (cycle0 (range_A))

act3                      (cycle0 (act_C))
range3                    (cycle0 (range_B))
day3                      (cycle13 (1))
```

Figure: V.3.6

As all the variables are unary, the program would announce success and stop (after propagation) If however a contradiction was reached during propagation *backup* would be initiated

Let us reflect on the example for a moment Compared to the first implementation, this program terminated very quickly By judiciously using the constraints, backtracking was reduced. Using constraints through mechanisms like *generation,posting and propagation*, search programs have been found to reduce backtracking dramatically (Stefik M. 1980).

V.4 Backtracking

To illustrate backtracking we now introduce the constraint C7 into the example we have been working on. C7 is a safety constraint that says " *doing activity: act_A on range_A on a particular day, will cause range_B to be unusable on that day.*" By glancing at the final results as shown in figure V.3.6 one notices that C7 is violated. On day '1' battalion: bat_A will be performing act_A on range_A, however range_B will be occupied by bat_B on the same day. This causes an error when constraint C7 is enforced.

There are several ways of backtracking at this point:

1: Change day3 to (2)

or 2: Change day1 to (2)

or 3: Change range1 to (range_B)

or 4: Some combination of the above

It is very difficult to decide upon which backtracking technique to adopt. To get around this decision, the current implementation just retracts the latest cycle. The latest cycle is cycle13 and retracting it is equivalent to adopting strategy 1 (above). By popping the stack for variable day3, the top of the stack now is: (cycle12 (2)). On propagating this, the program returns successfully.

Instead of ending on this rather encouraging note, we shall examine likely strategies for backtracking. Using chronological backtracking can often be very inefficient. There should be some way of finding out the best strategy. The first step in this direction is the identification of dependencies. In other words, we look for the culprits, the variables and constraints which cause the error to occur.

An error occurs when a constraint tries to set the value of a variable to the empty set (). The null set means that the variable got over-constrained and that the constraints have forced a contradiction to occur. At this point, the culprits are the corresponding constraints and all the variables that take effect through that constraint. As the constraints are the only form of domain

dependent knowledge, backtracking should be based on the form and structure of the constraints alone. I believe that the constraints can be pre-compiled into a complex, multi-referenced network. This symbolic constraint network would be able to *look-ahead* and make intelligent choices. An ability to look ahead is valuable because, under the current implementation, constraints seem to suddenly *pop* up when certain choices are made. The constraints, in some sense, lurk about the program and appear suddenly, often to the dismay of the scheduling program.

There is an important tradeoff while pre-compiling the constraints. It is possible that the time spent in developing the constraint network itself might waste too much time, one might be better of going ahead with the search.

Chapter VI

Future Directions

VI.1 Exploiting the Flexibility of CDL-II

VI.1.0 Personal Constraint sets

In *real world* scheduling problems there are several people involved in the development of the schedule. Each person has his/her own requirements off the schedule. As it is not possible to accommodate all the people, traditional scheduling programs tend to have a few, well established constraints *hard wired* into the system. These constraints, however, are subject to change. For example, changes in the staff of an organization can bring new managers who have unanticipated idiosyncrasies. You cannot rewrite old software to accommodate them!

Given the framework presented in this thesis, it is possible to input the constraints of several people at the same time. Each person will have his/her own set of constraints. The computer consequently tries to satisfy all the constraints. To facilitate usage, one would have to develop a higher level constraint definition language than the one (CDL-II) presented in chapter V. Let us call this natural constraint definition language: CDL-N.

Armed with something like CDL-N each person could input his/her own set of constraints. Further, he/she will be able to review and edit the constraint sets to suit his/her personal feelings. Each set of constraints will consequently reflect the personality of the person who owns it. These data sets can be added and removed when required, for example:- when a person gets transferred all he/she does is, take his/her constraint sets with him/her to his/her new job site. Likewise, if a senior manager retires, his/her *personality* can be retained in the form of CDL-N statements.

In addition to personality datasets there are datasets which correspond to other extraneous constraints:-

a) Personality dataset

b) Resources dataset

c) Shop floor constraints

d) Environmental factors

Environmental constraints can cover expected conditions like snow fall or financial climate, depending upon the application one is dealing with.

The system uses these constraints to draw up a plan and a schedule. It may not be possible to satisfy all the constraints. Under such conditions, the system will initially try ot satisfy the constraints which rank higher (fuzzy ranking). Further, a person higher up in the organizational structure will get higher weightages.

Drawing from the concept that an organization is basically an information processor (at some level), one will be able to model the whole organization. CDL-N could be an extension to the Business Definition Language developed by IBM corporation.

VI.1.2 What If Games

Once the constraint sets are entered, the users can go into a *what if* mode and can change their constraint sets to see how sensitive the system's response is to the changes he/she makes.

The system, in the process of scheduling should give reports on costs, resource requirement, performance standards etc. The user can play around with his data set and see how he can best adjust to the personalities of others.

The computer may even be able to ask itself what if questions.

VI.2 Handling Multiple Heuristics in Scheduling:

- towards a system Architecture

VI.2.0 Introduction

This section explores the techniques that may be used for handling multiple heuristics. Several researchers in Operations Research have developed heuristics for activity scheduling. Each of the heuristic is suitable for particular types of problems. None of the heuristics can perform well

in all scheduling problems. In this chapter a system architecture is proposed that which allows one to use these heuristics as and when required. It is stipulated that: If one could find out the conditions under which a particular heuristic is effective or ineffective, then it may be possible to recognize patterns and invoke the heuristics appropriately.

In section II.3 we introduced the concept of the scheduling cycle. In figure II.3.1 our cycle looped from the choice of 'who' to 'where' to 'what' to 'when' and back. There are two very fundamental questions that this formalism raises:

1) How does one decide upon the sequence of choices in the scheduling cycle.

2) Having generated some choices how does one choose which to pick.

There are no hard-and-fact algorithms or techniques by which these problems can be addressed. Only heuristic methods can be used to perform such tasks.

VI.2.1 Multiple Heuristics

Having decided to work with heuristics, how does one decide which kinds of heuristics are best for our problem. If we really do decide upon a particular heuristic, is it possible that half way through the scheduling process a different type of heuristic might be more relevant.

Heuristics come in all shapes and sizes. Aggressive strategies like to schedule as early as possible. "Wait & see" strategies exercise least-commitment. Backup heuristics help in undoing poor-choices.

Here are some examples:[Moder & Phillips 1983]

Name of Heuristic consultant	Description
Late Finish	Give priority to activities in order of increasing late finish time.
Minimum Slack	Schedule first those activities with low slack time.

Random	Priority given to jobs selected randomly
Bumping	If there is a clash, then bump the activity of lower priority.
Meta-OR	If total variables in the problem < 5000 & total constraints < 5000 then use Dynamic Programming

TABLE I

Given a set of heuristics we have to decide which one is most useful. Presumably different strategies are relevant in different situations. In addition some strategies may always performs better than other. There are two ways of invoking a heuristics

(a) Pattern directed

(b) Relative grading

Establishment of patterns for choosing a heuristic is very tough. The meta-OR heuristic in Table I is an example. We do not have any good ideas in this area yet.

Heuristics can be graded relatively. This is done by running the system on several typical problems, each time with only one of the heuristics in place. Performance characteristics of each heuristics is gathered and is used to rank the heuristics.

VI.2.2 Search Heuristics

When coming down the search tree we will have to adopt some kind of pruning mechanism. Having used some of the heuristics (like those in Table I) we reach a stage, at the end of a scheduling cycle, where there are several viable partial schedules. Due to time & computer memory constraints all these nodes cannot be developed. As mentioned in Sections II.2 & II.3, we need a function by which the nodes can be evaluated and then chosen for further branching.

From equation [3], section II.2, there are two parts of a evaluation function: (at node n)

$$f(n) = g(n) + h(n)$$

A measure of goodness of a node is based on utilities, equation [1] of section II.2 . This gives us g(n) only. The *estimate to complete* is found by doing a depth first search from the node in question. Such a search is designed to be a lower bound on the total utility and is hence conducted in a rash manner; constraints are not fully satisfied, no backtracking is performed, due dates are violated etc. This quick 'look ahead' will give us a good h(n) to work with. We now choose the node with highest f(n).

VI.3 System Architecture

Based on the ideas presented till now, we proceed to develop an architecture :

a) The constraints are defined by users and are subject to change.

b) The jobs (battalions) to be performed are ever-competing to be chosen next. For this reason they are said to exist in a *market of jobs*.

c) Each heuristic consultant has his own way of doing things they may either support one-another or give raise to conflicting situations. For this reason they are said to be in a *board of consultants*. These consultants communicate via a blackboard.

With this we are able to hone in on a system architecture. Figure VI.3.1 . The controller examines the advice (bids) deposited on the blackboard. A consultant is chosen and applied for a few scheduling cycles. The constraints are used as outlined in sections IV & V.

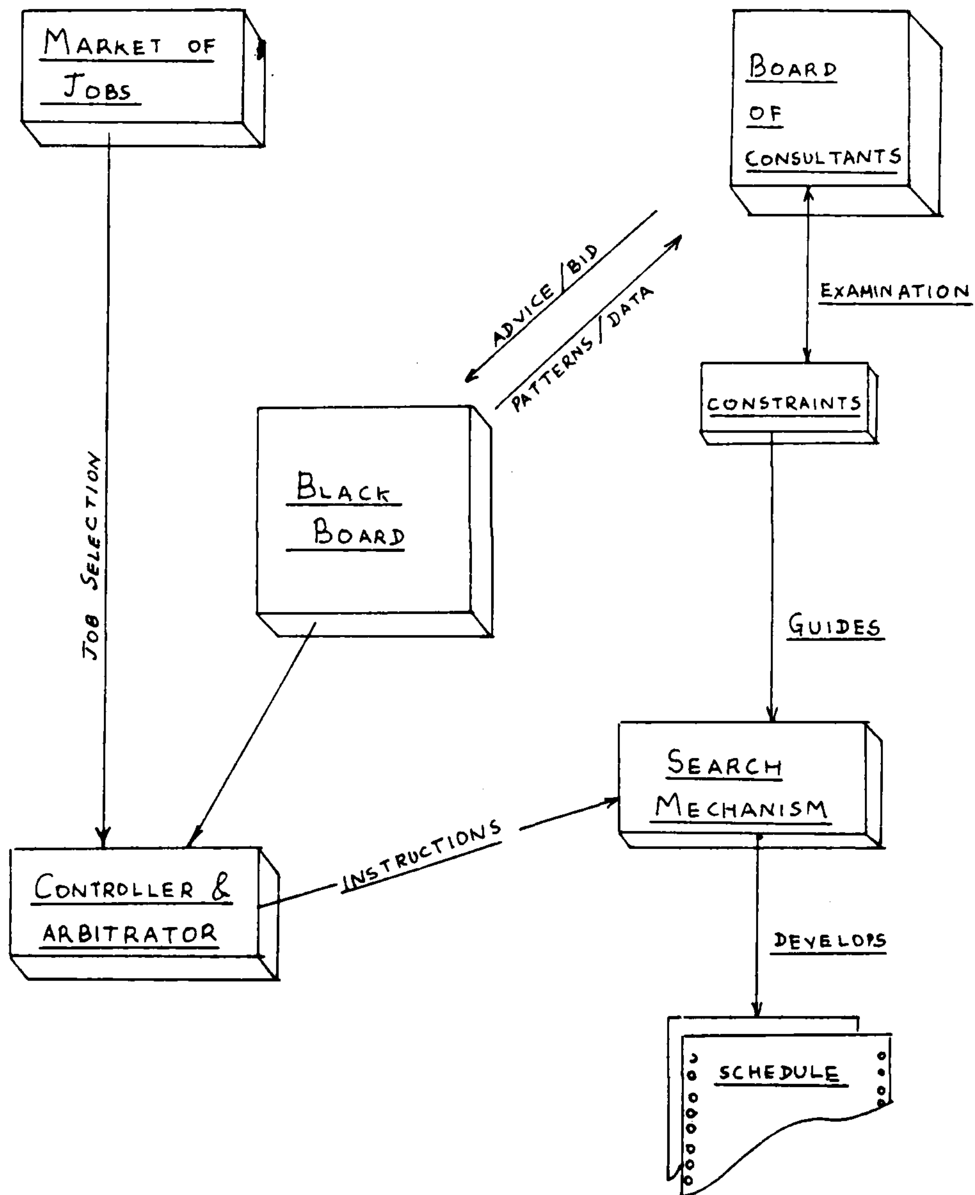


FIGURE VI.3.1 AN ARCHITECTURE

REFERENCES & BIBLIOGRAPHY

Bansal, S.P. 1977" Minimizing the Sum of Completion Times of a n Job over m machines in a Flowshop- A Branch&Bound Approach", AIIE Trans, Vol 9, No3, Sept 1977.

Chandra Navin, 1985 "IMST user's Manual: A tool for building Rule based Expert Systems" MIT, Center for Construction Research & Education, Technical Report: CCRE-85-6.

Doyle, J. 1979 "A truth maintenance system". AI 12: 231-272

Eilts, T.B., Wright, J.R., Houck, M.H. (1984) "The division gunnery Model (DIGUM) as an aid in Army training decision", Report CE-HSE-84-3, Purdue University.

Fahlman, Scott (1978) "A planning System for Robot Construction Tasks" (MS Thesis) AI TR-283 MIT AI Lab.

Fikes R.E. (1970) "REF-ARF: A system for Solving Problems Stated as Procedures", Artificial Intelligence, Vol1, pp27-120

Fikes R.E., and N.J. Nilsson (1971), "Strips: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence, Vol. 2, pp 189-208

Fox, M.S. (1983) "Constraint Directed Search: A case of Job Shop Scheduling". PhD Thesis, Carnegie-Mellon University.

Fukumori K., (1980) "Fundamental Scheme for train Scheduling", MIT AI Memo No 596, Artificial Intelligence Laboratory, MIT, Cambridge MA.

Goldstein I.P.,Robert R.B. (1977) "NUDGE: A knowledge-based Scheduling program," MIT AI Memo 405.

Moder JJ,C R Phillips, E W Davis (1983) "Project Management with CPM PERT and Precedence Diagramming", VNR Company NY

Nilsson N.J. (1971) "Problem solving Methods in AI", New York, N.Y.; Mc Graw Hill.

Ouciuch Ed, Frost John (1985)" ISA: Intelligent Scheduling [illegible]nt", AI Technology Center, Digital Equipment Corp. Hudson MA 01749

Sacerdoti, E.D. (1977) "A Structure for Plans and Behaviour." NY: [illegible]lsevier North-Holland , 1977. AI Series.

Stallman,R. and G.J. Sussman (1977) "Forward reasoning and dependency directed backtracking in a system for computer aided circuit-analysis", AI 9:135-196.

Stefik M. (1981) "Planning with Constraints (MOLGEN: Part1)", AI, Vol 16, pp 111-140.

Stefik M. (1980) "Planning with Constraints", STAN-CS-80-784, PhD. Thesis

Stinson Joel, David Edward,Khumwala Basheeer (1978). "Multiple Resource Constrained Scheduling using Branch & Bound", AIIE Trans, Vol 10, No 3, Sept 1978.

Sussman G.J., Steele G.L. (1980) "Constraints: A language for expressing almost heirarchical descriptions" AIJournal 14:1-39.

Tate, Austin (1977) "Generating Project Networks" IJCAI-77, Cambridge, Cambridge, MA 888-893

Vere, Steven A (1983) "Planning in Time. Windows and Durations for Activities and Goals." IEEE-PAMI, May 1983 pp245-267

Vere, Steven A (1984) "Temporal Scope of Assertions & Window cutoff." AI Research Group Report, Jet Propulsion Laboratory, Nov '84.

Wilkins, D.E. (1984) "Domain Independent Planning: Representation and Plan Generation." Artificial Intelligence. 22:3 April (1984). pp 269-301

Winston P.H., 1984 "Artificial Intelligence", 2nd Ed. , Addison-Wesely, Reading, Massachusetts.

APPENDIXA

IMST

Users Manual

A Tool for Building Rule-based Expert Systems

April 1985

Copyright © Navin Chandra

Center for Construction Research & Education
Department of Civil Engineering
Massachusetts Institute of Technology

ABSTRACT

IMST is a tool for building rule-based systems. It has been built to merge the technologies of Knowledge Representation in Artificial Intelligence with that of Relational Database Management systems. While maintaining a relational (in the eyes of the user) type database it allows the user to operate on the data using production rules.

IMST also allows the user to do some meta-level control. Rules can be loaded, and unloaded. Facts can be blackboarded for easy access and modularity.

IMST is written in Franz-LISP (with parts in C) and is built for the user who is not familiar with LISP. It, however, does allow the users to define their own functions and even their own interface.

This paper explains the workings of IMST through the use of numerous examples.

Table of Contents **Page**

1.0 Introduction

1.1 The System Architecture
1.2 The data base

2.0 An Example

2.1 Introduction
2.2 Example
2.3 More Examples

3.0 The rule Syntax

3.1 The rule
3.2 Test Predicates
3.3 Actions

4.0 Database Capabilities

4.1 Tabular data
4.2 Database actions

5.0 Miscellaneous Actions

5.1 Meta level Control
5.2 I/O Commands

6.0 Top-level Command Summary

7.0 IMST functions list

1.0 Introduction

IMST is an environment for building rule based expert systems. It is modeled after some of the popular production systems like OPS-5 (Charles Forgy '81 Carnegie-Mellon U.) and YAPS (Liz Allen '83 Maryland).

IMST was written to merge the technologies of rule based systems with that of data base management. It allows the user to think of his/her data in the form of a relational database.

The system is written in Franz-Lisp and it operates in a UNIX 4.2 environment. Even though IMST has its own top-level environment, it does not insulate the user from the full power of LISP.

1.1 The system architecture

IMST, like all other production systems, has two major parts; the rule base and a data base of facts. The rules operate upon the facts and make inferences. These inferences are added back to the facts list and are then available for use by the other rules.

IMST has its own top-level environment. The purpose of this environment is to let users interact with the system without knowing any LISP. Once an application is developed, the IMST top-level can be overridden by a user defined interface.

1.2 The data base

The data base is a collection of sentences. Each sentence is an assertion about the domain one is dealing with. Data can be entered into the database via the function **assert**.

The assertions are stored in a file which can be loaded into the IMST environment by using the **loadfile** command.

Consider a world about which we have the following information:

"There is a strong boy named john who lives on 33 Maple Street. There is a beautiful girl who is 5.5 ft tall. They both are good dancers."

This information can be broken up into assertional statements and can be stored in a file as

shown in figure 1.2.1 below.

```
(assert mary is_a girl)
(assert john is_a boy)
(assert john is strong)
(assert john address 33_maple_street)
(assert john is a good dancer)
(assert mary is beautiful)
(assert mary height 5.5)
(assert mary is a good dancer)
```

Figure: 1.2.1 ---

The database of facts can now be queried and can be used by a rule base.

Whenever an assertion is made, the first word is treated as an object and all subsequent assertions having the same first word are grouped together. The assertions listed above will be stored in the database like this:

```
john:                                   mary:

(john is_a boy)                         (mary is_a girl)
(john address 33_maple_street)          (mary is beautiful)
(john is a good dancer)                 (mary height 5.5)
(john is strong)                        (mary is a good dancer)
```

Figure: 1.2.2 ---

The reader should be aware that there is no restriction on the contents of an assertion. The same data could be asserted like this:

```
(assert there is a boy named john)
(assert he is strong and he lives on 33_maple_street)
(assert john is a good dancer)
(assert she is mary)
(assert she dances well)
(assert mary is five feet 6 inches)
```

```
(assert she is beautiful)
```

Figure: 1.2.3 ———————————————————————

The semantic content is the same but there is no modularity and *consistency*. The facts (mary dances well) and (john is a good dancer) may be the same for the user but not to the computer. The use of statements like (she is beautiful) can be very misleading. It is advisable to use the (<object> <attribute> <value>) format . In essence, consistency is highly desirable.

Before closing this section it is useful to know that a database fact can be removed by the **unassert** *function*. The *file:*

———————————————————————

```
(unassert john is strong)
(unassert john is_a boy)
(unassert john is a good dancer)
(unassert john address 33_maple_street)
```

Figure 1.2.4 ———————————————————————

will, when loaded, delete all the information on john.

2.0 An example

2.1 Introduction

This chapter is intended to instruct the user about the use of IMST. It contains a simple example and attempts to explain the important features of IMST through the example.

In section 1.2 the notion of an assertional database was introduced. We now show how the data may be used by production rules.

A production rule is basically an IF-THEN rule. A rule states that IF certain facts are true THEN there are a few other facts that are deemed true.

The example below is a fully annotated trace of the IMST environment. The text in **boldface** is which is typed in by the user. The text in *italics* is the explanation and the normal typeface is that which is printed by the computer.

2.2 Example

2.2.1 First the data:

To start IMST one first logs into the UNIX system where the program has been installed. After logging in just type 'imst' to the unix shell

unix% **imst**

This may be followed by a few system messages and in about 5-8 seconds you will find yourself with the IMST prompt: 'imst> '. The first thing to do is to initialize the system with the 'init' command

imst> **init**

Now let us input some data. The data is normally added to the system via a file. To create such a file we can go into the editor by using the 'emacs' command. Let us call the file 'people'

imst> **emacs people**

This will put you into a full screen editor. If you are not familiar with emacs you could use the

command: 'vi' (for the unix visual editor). Here is how the data is entered in the file.

```
(assert john is_a boy)
(assert john height 6.0)
(assert john gpa 3.8)
(assert john likes sailing)

(assert jack is_a boy)
(assert jack height 5.5)
(assert jack gpa 5.0)

(assert mary is_a girl)
(assert mary is beautiful)
(assert mary gpa 4.2)
(assert mary height 5.5)
(assert mary likes sailing)

(assert boy is_a person)
(assert girl is_a person)
(assert person is_a mortal)
(assert person has a soul)
```

Figure: 2.2.1 --

Having typed the data in, exit emacs normally. This will put you back in IMST. You are now ready to load the file.

```
imst> loadfile people
```

The above command will produce the following data base in IMST

```
john:  (john is_a boy)          jack:  (jack is_a boy)
       (john height 6.0)               (jack height 5.5)
       (john gpa 3.8)                  (jack gpa 5.0)
       (john likes sailing)

mary:  (mary is_a girl)         girl:  (girl is_a person)
       (mary is beautiful)
       (mary gpa 5.0)           boy:   (boy is_a person)
       (mary height 5.5)
       (mary likes sailing)     person: (person is_a mortal)
                                        (person has a soul)
```

Figure 2.2.2 --

There is a way of viewing this data. This is shown below:

imst> describe john

```
---------------------------------------------------------------------------

        Description of object: john

                is_a    boy
                height   6.0
                gpa   3.8

---------------------------------------------------------------------------

        imst>
```

2.2.2 Now for some rules

A rule consists of a set of facts (patterns) in the IF part, followed by a set of actions in the THEN part. The IF & THEN parts are separated by a implication sign '-->'

Here is the syntax:

(rule -name- -pattern_1- -pattern_2-

.... -pattern_n-

--> -action_1- -action_2- .... -action_m-)

Assume that we want to conclude that john knows how to swim because we know that he likes sailing. Here is what the rule may look like :

(rule john_swim (john likes sailing)

--> (assert john knows swimming))

There is only one pattern in this rule and when the rule is run, it will fire because the pattern (john likes sailing) is found to exist in the database. The action 'assert' will add (john knows swimming) to the database.

There is a problem here, we also want to conclude that mary can swim because she too goes sailing. The rule should be able to handle all those who like sailing. In other words the rule should be able to handle variables. Here is what it might say "IF there is an 'x' which likes sailing, then assert

that that 'x' knows swimming"

This is done by introducing a variable into the pattern. The pattern (>x likes sailing) will match any fact which has any object with the words 'likes' & 'sailing' in the attribute and value positions. In other words, (>x likes sailing) will match the fact (john likes sailing) and (mary likes sailing) and will bind x to john & mary respectively.

Assuming x is bound to 'john', the assertion (assert <x knows swimming) will actually assert (john knows swimming). The use of the the symbol > before a variable name means "bind this variable" & the symbol < means "lookup the value ".

A mnemonic way of looking at this is:

>x can be looked upon as -->x , that is, a value going into x

and <x can be looked upon as <--x , that is, the value coming out of the variable x

Let us now write the rule in a file called 'rule'

```
(rule   swimming_rule   (>x likes sailing)

    -->    (assert <x knows swimming))
```

```
    imst>    loadfile rule

    imst>    run
Trying rule: swimming_rule

        asserting > (john knows swimming)
        asserting > (mary knows swimming)

done

        imst> describe john
--------------------------------------------------------------------------
        Description of object john

           is_a boy
           height  6.0
           gpa  3.8
```

```
        likes sailing
        knows swimming
```

Notice that the data for object john has been updated. We now go on to write some more rules:

Here is a rule that says that any boy with a gpa of 5.0 is a nerd (we have a 5.0 system for Grade point average here at MIT).

```
(rule nerd (>x gpa 5.0
        (<x is_a boy)

->      (assert <x is_a nerd))
```

Here is how the rule works:

The first pattern will match (jack gpa 5.0) and x will be bound to 'jack'. The second pattern has a lookup for x and will become (jack is_a boy). As the second pattern is found in the data base, the rule will fire and (jack is_a nerd) will be asserted. Notice that the rule will not fire for mary.

2.2.3 More examples

a) "If any boy's height is greater than 6 feet, then he is tall"

We now introduce yet another part of the IMST rule: *the test.* A test is a function that either returns true or false.

```
(rule tall_boys (>a1 is_a boy)
                (<a1 height >ht)
        test    (ge <ht 6.0)

-->     (assert <a1 is tall) )
```

The rule says: "IF there is a boy 'a1' of height 'ht' and if 'ht' is greater-than-or-equal-to (ge) 6.0 then assert that 'a1' is tall."

b) "If any boy is less than 6.0 feet tall but greater than 5.5 feet in height, then that person is of average height"

```
(rule   average_height_boys
        (>x is_a boy)
        (<x height >ht)
test    (ge <ht 5.5)
        (lt <ht 6.0)

--> (assert <x is of_average_height))
```

c) "All beautiful girls like nerds"

```
(rule   only_at_mit
                (>x is_a girl)
                (<x is beautiful)
                (>y is_a nerd)

-->     (assert <x likes <y))
```

Notice how this rule uses the inference made by rule "nerd" of section 2.2.2

d) "To find who is taller than whom"

```
(rule taller (>x height >xht)
             (>y height >yht)
        test (gt <xht <yht)

-->     (assert <x is taller than <y))
```

e) Now an interesting rule on inheritance:

Inheritance is an important part of building object based semantic networks. If an object 'y' is_a object 'x' (e.g. dog is_a animal), then the object 'y' should inherit the propertis of 'y'. Stated in rule form we have:

"If there is any object x with property px and value vx and if some y is_a x then y should also have the property px and value vx"

```
(rule inheritance      (>y is_a >x)
                       (<x >px >vx)
        -->     (assert <y <px <vx))
```

If we run this rule on the data in 2.2.2 we will get the following assertions

---

```
(jack is_a person)  (jack is_a mortal)  (jack has a soul)
(mary is_a person)  (mary is_a mortal)  (mary has a soul)
(john is_a person)  (john is_a mortal)  (john has a soul)
```

You have just been exposed to the most important part of IMST and are ready to write rule based systems!

f) A arithmetic rule: To convert the height to meters.

```
        (rule feet_to_meters
        (>x height >y)
-->     (unassert <x height <y)
        (assert <x height_metric
                (/ (* (* <y 12.0) 2.54) 100.0) ))
```

3.0 The rule Syntax

3.1 The rule

The rule in IMST are of the following form

```
(rule  -rulename-     -pattern_1-
                      -pattern_2-
                ..... -pattern_n-

       test     -test_1-    -test_2-
                      ...... -test_p-

-->    -action_1-   -action_2- ...... -action_m- )
```

The use of tests is optional. If there are no tests, simply drop the word 'test' from the body of the rule.

The rule fires when all the patterns match & all the tests are true. The tests are evaluated *using the lisp eval, the same goes for the actions.*

3.2 Test Predicates

There are several predicates that are provided by IMST:

(ge -arg1- -arg2-) Returns true if arg1 is greater than or equal to arg2.

(le -arg1- -arg2-) Less than or equal to

(gt -arg1- -arg2-) Greater than.

(lt -arg1- -arg2-) Less than.

(= -arg1- -arg2-) Equal to

(ne -arg1- -arg2-) Not Equal to

(not* -pattern1-) The pattern1 does not exist in the database.(The pattern should not have any variables in it.)

3.3 Actions

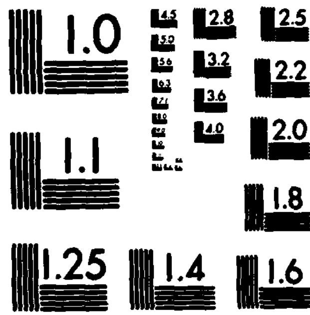
The most common action is **assert**. One can use any user-defined LISP function. By using Franz Lisp's Fasl function it is possible to include actions that are written in C or Fortran.

The other pre-defined actions provided by IMST are presented in the following chapters.

AD-A171 514 INTELLIGENT USE OF CONSTRAINTS FOR ACTIVITY SCHEDULING 2/2
(U) CONSTRUCTION ENGINEERING RESEARCH LAB (ARMY)
CHAMPAIGN IL NAVINCHANDRA AUG 86 CERL-TM-N-86/15

UNCLASSIFIED F/G 12/2 NL

END
DATE
FILMED
10-86
DTI



:CROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

4.0 Database Capabilities

In building real-life systems, it is not possible to have a long data file full of assertions. IMST has a ability to handle tabular data.

4.1 Tabular Data

4.1.1

It is popular to think of data in a tabular form. IMST allows you to input data in this form. Consider the table below

student	ID#	Gpa	Height
john	43433	3.8	6.0
jack	39393	5.0	5.5
mary	39204	5.0	5.5

The data is converted into object-attribute-value triples automatically . Here is what will be asserted for john:

```
(john ID# 43433)
(john gpa 3.8)
(john height 6.0)
(john is_a student)
```

The above 'is_a' relationships are implied by the virtue of the fact that the names john, jack & mary are listed below the field 'student'.

4.1.2 The **create, edit & loaddata** commands.

create

The create command takes no arguments and is used to create a table. Here is a self explanatory example:

```
imst> create

Name of the file =>  people
        type 'quit' to finish up

Field 1 => student
Field 2 => ID#
Field 3 => gpa
Field 4 => height
Field 5 => quit

imst>
```

A file should be created only once. Having created the file, it is now ready for editing.

edit

edit takes no arguments and it is used to edit a tabular data file. It allows one to input data record by record. It can be used to edit old files too.

The editor is currently very simple, it only allows you to go forward from field to field & from record to record. After typing in data one can exit the editor by hitting return on a blank line only. If you make a mistake, you will have to exit the editor and re-enter the editing routine, you cannot go back.

Note: The editor forces you to use a directory to store your files. You can choose your current directory by providing the appropriate pathname in full.

For example: If we wish to store the above file 'people' in a sub-directory called 'knowledge-base', then the trace may look like this:

```
imst> edit

Give me a directory name =>  knowledge-base

Name of the data file within the above directory => people
 Which record would you like to start at?
 (if  this is a new file, start at #1) => 1
```

loaddata

This function takes no arguments and it is used to load in a file which was a product of the **create** and **edit** commands. It automatically converts the records into a frame representation and makes the appropriate assertions.

4.2 Database actions

Here are some of the database actions that IMST can perform within the body of a rule:

(**assert** ...data...) To assert data into the database.

(**unassert** ....data...) To remove a data element from the database.

(**glue** string_1 string_2) It is used to create new objects. For example: If x is bound to 'john' and we wish to create a new object called 'johns_girlfriends' then one could use:

(assert (glue <x s_girlfriends) ....data....)

(**cardinality** object1) Counts the number of facts within a given object1.

(**summation** field1 object1) Returns the summation of a particular field from an object. It requires that the selected object has the specified in the 'value' position (rightmost) in each fact.

(**average** field1 object1) Same as summation, only it returns the arithmetic mean .

5.0 Miscellaneous Actions

5.1 Meta Level Control

The rule base in IMST exists in two areas, the active and the inactive set.

(**swapin** rule1) This will swapin the rule1 into the active set from the inactive set.

(**swapout** rule1) Will send an active rule to the inactive set.

(**rmrule** rule1) Will remove the rule1 from the active set, forever.

(**loadfile** file_1) This will load the file IMST into the top-level and is useful for pattern invoked loading or rulebases.

(**initrules**) Will clean up all the rules in the current active ruleset.

5.2 I/O Commands

(**skip** n) Skips 'n' lines.

(**tab** n) Will tab cursor forward by n columns.

(**say** ...message....) The say action is like assert but does not assert the message but just echo's it to the screen.

(**ask** ...question....) It will accept a value for a variable in the text of the question. For example:

(ask What is your >name)

The system will stop and wait for input from the user. The input will be bound to the variable 'name' .

6.0 Top-Level Command Summary

A summary of all the commands that can be issued from the IMST top-level.

create Creates table

describe object Returns a description

dumplisp filename Will dump the whole system, the data, the rules and IMST into the specified file. You need about 2.0 megabytes.

edit Edit a created table

emacs file Edit a unix file using emacs.

help Gives some help

init Initializes the whole system.

initrules Throws away the current *ruleset but not the* data.

lisp Lets you talk to Franz lisp directly.

loaddata Used for tabular data loading.

loadfile filename Will load the specified file.

quit Quit

run Starts the inference Engine

top Return to top-level of IMST.

vi file The unix vi editor

7.0 IMST function list

The test and actions in IMST can be any lisp function. IMST comes with a library of useful functions:

Arithmetic functions:

The operators: + / * -

The predicates: gt lt ge le = ne not*

Database operations:

assert

unassert

glue

cardinality

summation average

I/O

say

ask

skip

tab

Meta-level Control

swapin

swapout

rmrule

loadfile

loaddata

initrules

APPENDIXB

Trace of a run

This appendix is the trace of a run. It has three part:

PartI The input data to the program.

PartII The constraints (in CDL-II)

PartIII An annotated trace of the run.

The example used here is an extention of the case presented in Chapter I.

Appendix B

Part I

The input data

```
;;T H E    I N P U T   D A T A
;;

;; This file contains the data used for the trace of CDL-II
;; It contains the definition of all the schedule elements
;;


;;****************************************************************
;; The TIME PERIODS & FIRING RANGES
;;****************************************************************

;; Setting the full time period to be one month

(set 'one_month '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
                  19 20 21 22 23 24 25 26 27 28 29 30 ))

;; Setting the holidays

(set 'holidays '(1 6 14 22 28))

;; Setting sub time periods: early and late.

(set 'early  '(1 2 3 4 5 6))

(set 'late  '(26 27 28 29 30))

;; Setting the firing ranges

(set 'all_ranges '(range_A  range_B   range_C  range_D  range_E range_F))


;;****************************************************************
;; THE BATTLIONS
;; Defining the battalions and their schedule elements
;;****************************************************************


;;-----------------------------------------------------------
;; BATTALION : bat_A

;; The first schedule element, "bat_A will carry out
;; act_A on any range any time of the total period

(assert bat_A actA1 rangeA1 dayA1)

(set_value 'actA1 '(act_A))
(set_value 'dayA1 one_month)
(set_value 'rangeA1 '(range_A  range_B  range_C))

;; The second schedule element:

(assert bat_A actA2 rangeA2 dayA2)

(set_value 'dayA2 one_month)
(set_value 'rangeA2 '(range_A range_B range_C))
(set_value 'actA2 '(act_A))

;; The third schedule element

(assert bat_A  actA3 rangeA3 dayA3)

(set_value 'dayA3 one_month)
(set_value 'rangeA3 '(range_A range_B range_C))
(set_value 'actA3 '(act_A))
```

```
;; The fourth schedule element

(assert bat_A actA4 rangeA4 dayA4)

(set_value 'actA4 '(act_B))
(set_value 'rangeA4 all_ranges)
(set_value 'dayA4 early)

;;
;;-----------------------------------------------------------------


;;-----------------------------------------------------------------
;; BATTALION 'B'

;; First Schedule element:
(assert bat_B actB1 rangeB1 dayB1)

(set_value 'dayB1 one_month)
(set_value 'rangeB1  all_ranges)
(set_value 'actB1 '(act_B))

;; the second schedule element:
(assert bat_B actB2 rangeB2  dayB2)

(set_value 'actB2 '(act_A))
(set_value 'rangeB2 all_ranges)
(set_value 'dayB2 late)

;; the third schedule element:

(assert bat_B actB3 rangeB3 dayB3)

(set_value 'actB3 '(act_C))
(set_value 'rangeB3 all_ranges)
(set_value 'dayB3 late)


;;-----------------------------------------------------------------

;; BATTALION "C"

; the first element

(assert bat_C actC1 rangeC1 dayC1)

(set_value 'actC1 '(act_A))
(set_value 'rangeC1 all_ranges)
(set_value 'dayC1 one_month)

;; the second element

(assert bat_C actC2 rangeC2 dayC2)

(set_value 'actC2 '(act_B))
(set_value 'rangeC2 all_ranges)
(set_value 'dayC2 early)

;; the third elememnt

(assert bat_C actC3 rangeC3 dayC3)

(set_value 'actC3 '(act_B))
(set_value 'rangeC3 all_ranges)
```

```
(set_value 'dayC3 one_month)

;;----------------------------------------------------------------

;; BATTALION "D"

; the first element

(assert bat_D actD1 rangeD1 dayD1)

(set_value 'actD1 '(act_A))
(set_value 'rangeD1 all_ranges)
(set_value 'dayD1 early)


;; the second element

(assert bat_D actD2 rangeD2 dayD2)

(set_value 'actD2 '(act_A))
(set_value 'rangeD2 all_ranges)
(set_value 'dayD2 late)
```

Appendix B

Part II

The Constraints (in CDL-II)

```
;; THE        C O N S T R A I N T S    ( IN    CDL-II)
;;
;; The constraint set presented here is used in the attached trace of
;; the program.
;;
;;-----------------------------------------------------------------------
;;

;;**********************************************************************
;; CONSTRAINT C1
;;
;; A battalion can do only one activity per day

(constraint C1 (>bat >act >range >day)
          test (selected? <day)

--> (constraint c1_aux (<bat >a1 >r1 >d1)

        test (not (equal (quote <day)  d1))

        --> (set_value  d1
                  (subtract (get_value d1)
                        (get_value (quote <day))))
          ))

;;*******************************************************************


;;********************************************************************
;; CONSTRAINT C2
;;
; Constraint 2:: Two battalions cannot be on the same range
; on the same day.

(constraint C2 (>bat >act >range >day)
  test (selected? <range <day)

--> (constraint  C2_one (>b2 >a2 >r2 >d2)
       test (not (equal a2 (quote <act)))  ; make sure its not the same one!
           (equal (get_value r2) (quote ! (get_value range)))

           --> (set_value d2
                (subtract (get_value d2) (quote ! (get_value day)))))

   (constraint C2_two (>b3 >a3 >r3 >d3)
       test (not (equal a3 (quote <act)))
           (equal (get_value d3) (quote ! (get_value day)))

           --> (set_value r3
                (subtract (get_value r3) (quote ! (get_value range))))) )

;;**********************************************************************


;;**********************************************************************
;; CONSTRAINT C3
;;
;; constraint C3
;
; if any battalion is scheduled for activity act_B then that
; battalion should not be scheduled for anything
; the very next day.
```

```
(constraint C3
      (>bat >act >range >day)
test   (selected? <day <act)
      (equal (get_value act) '(act_B))

--> (constraint C3_aux (<bat >a1 >r1 >d1)
      test (not (equal (quote <day ) d1))

   --> (set_value d1
      (subtract (get_value d1)
              (list (+ 1 (car (get_value (quote <day))))))) ))

;;*******************************************************************


;;*******************************************************************
;; CONSTRAINT C4
;;
; Constraint C4
; assignment of ranges to activities
;
; actvity      acceptible ranges
;
;  act_A        range_A range_B
;  act_B        range_C
;  act_C        range_B
;

(constraint C4_one (>bat >act >range >day)
         test (equal (get_value act) '(act_A))
         --> (set_value range '(range_A range_B)))

(constraint C4_two (>bat >act >range >day)
         test (equal (get_value act) '(act_B))
         --> (set_value range '(range_C)))

(constraint C4_three (>bat >act >range >day)
         test (equal (get_value act) '(act_C))
         -->  (set_value range '(range_B)))

;;*******************************************************************


;;*******************************************************************
;; CONSTRAINT C6
;;
;;
;; Cyclic constraint on activity act_A.
;; the window is set at x+5 and x+10



(constraint C6_one (bat_A  actA1 rangeA1 dayA1)
                          ; being specific to battalion_one
                          ; could be done automatically .

        --> (set_value 'dayA3 (restrict (get_value 'dayA3)
                    '(and (ge $$$ (+ (get_min_value 'dayA2) 5))
                         (le $$$ (+ (get_max_value 'dayA2) 10)))))

          (set_value 'dayA2 (restrict (get_value 'dayA2)
                    '(and (ge $$$ (+ (get_min_value 'dayA1) 5))
                         (le $$$ (+ (get_max_value 'dayA1) 10))
                         (le $$$ (- (get_max_value 'dayA3) 5))
                         (ge $$$ (- (get_min_value 'dayA3) 10)))))
```

```
        (set_value 'dayA1 (restrict (get_value 'dayA1)
                    '(and (le $$$ (- (get_max_value 'dayA2) 5))
                         (ge $$$ (- (get_min_value 'dayA2) 10)))))
        )

;;*****************************************************************
;; CONSTRIANT C7
;;
;;Constriant C7
;;
;; The safety spans constraint: When act_A is carried out on range_A
;; then the safety spans requires that it is unsafe to schedule anything on
;; range_C


(constraint C7_safety_spans (>bat >act >range >day)

        test (selected? <day <act <range)
            (equal (get_value act) '( act_A))
            (equal (get_value range) '( range_A))

;; has two parts: If the range is range_C then do not use the day
;;              If the day is the same then do not use range_C

--> (constraint C7_aux (>bb >aa >rr >dd)
            test (not (equal aa (quote <act)))
                (equal (get_value rr) '(range_C))
            --> (set_value dd
                (subtract (get_value dd)
                        (get_value (quote <day)))))

  (constraint C7_aux_aux (>bbb >aaa >rrr >ddd)
            test (not (equal aaa (quote <act)))
                (equal (get_value ddd) (quote <day))

            -->   (set_value rrr
                  (subtract (get_value rrr)
                          '(range_C)))))


;;;;
;;*****************************************************************


;;*****************************************************************
;;
;; CONSTRAINT C8
;;
;;
;;; Constraint C8 : "there shall be no training on holidays"
;;;

(constraint C8 (>bat >act >range >day)
        -->

        (set_value day
              (restrict (get_value day)
                      '(not (member $$$ holidays)))))

;;*****************************************************************
```

Appendix B

Part III

The Trace of the Program

This trace uses the data and the constraints shown in parts I and II of this appendix. The program is started by entering lisp, loading the program, the data, and then issuing the command : (search)

The program follows the algorithm oulined in section V.3 of this thesis. The main steps are:

(1) Try to propagate the constraints.

Whenever a constraint causes an effect
the new value is echoed along with the
current cycle number.

At the end of each propagate cycle the
program echoes "PROPAGATION COMPLETE"

(2) After the above step, it branches and sets up
new contexts. The word "BRANCHING" is echoed
on the screen.

If during the propagation stage the program encounters a contradition, it backs up and "PURGES" the latest context.

```
Script started on Wed May 8 10:43:31 1986
hera% rsh hades lisp
Franz Lisp, Opus 38.91


-> (load 'RUN.l)
[load RUN.l]
[fasl compiled/rulengine.o]
[fasl compiled/cdl.o]
[fasl compiled/utils.o]
[fasl compiled/search.o]

you will have to load data and then issue command (search)t

-> (load 'data.l)
[load data.l]
  asserting > (bat_A actA1 rangeA1 dayA1)
 Setting value of actA1 to (act_A) with context = cycle

 Setting value of dayA1 to (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30) with conte

 Setting value of rangeA1 to (range_A range_B range_C) with context = cycle

  asserting > (bat_A actA2 rangeA2 dayA2)
 Setting value of dayA2 to (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30) with conte

 Setting value of rangeA2 to (range_A range_B range_C) with context = cycle

 Setting value of actA2 to (act_A) with context = cycle

  asserting > (bat_A actA3 rangeA3 dayA3)
 Setting value of dayA3 to (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30) with conte

 Setting value of rangeA3 to (range_A range_B range_C) with context = cycle

 Setting value of actA3 to (act_A) with context = cycle

  asserting > (bat_A actA4 rangeA4 dayA4)
 Setting value of actA4 to (act_B) with context = cycle

 Setting value of rangeA4 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayA4 to (1 2 3 4 5 6) with context = cycle

  asserting > (bat_B actB1 rangeB1 dayB1)
 Setting value of dayB1 to (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30) with conte

 Setting value of rangeB1 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of actB1 to (act_B) with context = cycle

  asserting > (bat_B actB2 rangeB2 dayB2)
 Setting value of actB2 to (act_A) with context = cycle

 Setting value of rangeB2 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayB2 to (26 27 28 29 30) with context = cycle

  asserting > (bat_B actB3 rangeB3 dayB3)
 Setting value of actB3 to (act_C) with context = cycle

 Setting value of rangeB3 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayB3 to (26 27 28 29 30) with context = cycle
```

```
  asserting > (bat_C actC1 rangeC1 dayC1)
 Setting value of actC1 to (act_A) with context = cycle

 Setting value of rangeC1 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayC1 to (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30) with conte

  asserting > (bat_C actC2 rangeC2 dayC2)
 Setting value of actC2 to (act_B) with context = cycle

 Setting value of rangeC2 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayC2 to (1 2 3 4 5 6) with context = cycle

  asserting > (bat_C actC3 rangeC3 dayC3)
 Setting value of actC3 to (act_B) with context = cycle

 Setting value of rangeC3 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayC3 to (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30) with conte

  asserting > (bat_D actD1 rangeD1 dayD1)
 Setting value of actD1 to (act_A) with context = cycle

 Setting value of rangeD1 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayD1 to (1 2 3 4 5 6) with context = cycle

  asserting > (bat_D actD2 rangeD2 dayD2)
 Setting value of actD2 to (act_A) with context = cycle

 Setting value of rangeD2 to (range_A range_B range_C range_D range_E range_F) with context = cycle

 Setting value of dayD2 to (26 27 28 29 30) with context = cycle
```

We can inquire the database by using the describe function. This function takes as an argument the name of an object. It echoes all the associated variables and their values.

```
->

(describe 'bat_A)
actA4 = (act_B)    cycle = cycle
rangeA4 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayA4 = (1 2 3 4 5 6)    cycle = cycle
actA3 = (act_A)    cycle = cycle
rangeA3 = (range_A range_B range_C)    cycle = cycle
dayA3 = (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)    cycle = cycle
actA2 = (act_A)    cycle = cycle
rangeA2 = (range_A range_B range_C)    cycle = cycle
dayA2 = (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)    cycle = cycle
actA1 = (act_A)    cycle = cycle
rangeA1 = (range_A range_B range_C)    cycle = cycle
dayA1 = (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)    cycle = cycle
nil
-> (describe 'bat_B)
actB3 = (act_C)    cycle = cycle
rangeB3 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayB3 = (26 27 28 29 30)    cycle = cycle
actB2 = (act_A)    cycle = cycle
```

```
rangeB2 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayB2 = (26 27 28 29 30)    cycle = cycle
actB1 = (act_B)    cycle = cycle
rangeB1 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayB1 = (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)    cycle = cycle


-> (describe 'bat_C)
actC3 = (act_B)    cycle = cycle
rangeC3 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayC3 = (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)    cycle = cycle
actC2 = (act_B)    cycle = cycle
rangeC2 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayC2 = (1 2 3 4 5 6)    cycle = cycle
actC1 = (act_A)    cycle = cycle
rangeC1 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayC1 = (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)    cycle = cycle
nil
-> (describe 'bat_D)
actD2 = (act_A)    cycle = cycle
rangeD2 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayD2 = (26 27 28 29 30)    cycle = cycle
actD1 = (act_A)    cycle = cycle
rangeD1 = (range_A range_B range_C range_D range_E range_F)    cycle = cycle
dayD1 = (1 2 3 4 5 6)    cycle = cycle
nil
```

Having loaded all the data we now issue the command: (search) We do not have to load the constraints yet, the search routine will do that automatically.

The computer starts off in the propagation state. Whenever a constraint is used it echoes the fact that it is setting the value of a variable to some new value set. The computer echos "USING CONSTRAINT:" followed by the name of the constraint.

```
-> (search)
[load constraints.l]
USING CONSTRAINT: C8

Setting value of dayD2 to (26 27 29 30) with context = cycle

Setting value of dayD1 to (2 3 4 5) with context = cycle

Setting value of dayC3 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

Setting value of dayC2 to (2 3 4 5) with context = cycle

Setting value of dayC1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

Setting value of dayB3 to (26 27 29 30) with context = cycle

Setting value of dayB2 to (26 27 29 30) with context = cycle

Setting value of dayB1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

Setting value of dayA4 to (2 3 4 5) with context = cycle
```

Setting value of dayA3 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

Setting value of dayA2 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

Setting value of dayA1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

USING CONSTRAINT: C6_one

Setting value of dayA3 to (7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

Setting value of dayA2 to (7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25) with context = cycle

Setting value of dayA1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20) with context = cycle

USING CONSTRAINT: C4_three

Setting value of rangeB3 to (range_B) with context = cycle

USING CONSTRAINT: C4_two

Setting value of rangeC3 to (range_C) with context = cycle

Setting value of rangeC2 to (range_C) with context = cycle

Setting value of rangeB1 to (range_C) with context = cycle

Setting value of rangeA4 to (range_C) with context = cycle

USING CONSTRAINT: C4_one

Setting value of rangeD2 to (range_A range_B) with context = cycle

Setting value of rangeD1 to (range_A range_B) with context = cycle

Setting value of rangeC1 to (range_A range_B) with context = cycle

Setting value of rangeB2 to (range_A range_B) with context = cycle

Setting value of rangeA3 to (range_A range_B) with context = cycle

Setting value of rangeA2 to (range_A range_B) with context = cycle

Setting value of rangeA1 to (range_A range_B) with context = cycle

USING CONSTRAINT: C6_one

Setting value of dayA3 to (12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30) with context = cycle

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of rangeD2 to (range_B) with context = cycle0

Setting the value of rangeD2 to (range_A) with context = cycle1

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of rangeD1 to (range_B) with context = cycle2

Setting the value of rangeD1 to (range_A) with context = cycle3

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of rangeC1 to (range_B) with context = cycle4

Setting the value of rangeC1 to (range_A) with context = cycle5

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of rangeB2 to (range_B) with context = cycle6

Setting the value of rangeB2 to (range_A) with context = cycle7

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of rangeA3 to (range_B) with context = cycle8

Setting the value of rangeA3 to (range_A) with context = cycle9

BRANCHING ...

Setting the value of rangeA2 to (range_B) with context = cycle10

Setting the value of rangeA2 to (range_A) with context = cycle11

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of rangeA1 to (range_B) with context = cycle12

Setting the value of rangeA1 to (range_A) with context = cycle13

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayD2 to (30) with context = cycle14

Setting the value of dayD2 to (29) with context = cycle15

Setting the value of dayD2 to (27) with context = cycle16

Setting the value of dayD2 to (26) with context = cycle17

Setting value of dayC1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 27 29 30) with context = cycle17

Setting value of dayB2 to (27 29 30) with context = cycle17

Setting value of dayA3 to (12 13 15 16 17 18 19 20 21 23 24 25 27 29 30) with context = cycle17

USING CONSTRAINT: C7_aux_aux

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 27 29 30) with context = cycle17

Setting value of dayB1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 27 29 30) with context = cycle17

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayB2 to (30) with context = cycle18

Setting the value of dayB2 to (29) with context = cycle19

Setting the value of dayB2 to (27) with context = cycle20

USING CONSTRAINT: c1_aux

Setting value of dayB3 to (26 29 30) with context = cycle20

Setting value of dayB1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle20

USING CONSTRAINT: C2_one

Setting value of dayC1 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle20

Setting value of dayA3 to (12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle20

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle20

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayB3 to (30) with context = cycle21

Setting the value of dayB3 to (29) with context = cycle22

Setting the value of dayB3 to (26) with context = cycle23

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayD1 to (5) with context = cycle24

Setting the value of dayD1 to (4) with context = cycle25

Setting the value of dayD1 to (3) with context = cycle26

Setting the value of dayD1 to (2) with context = cycle27

USING CONSTRAINT: C2_one

Setting value of dayC1 to (3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle27

Setting value of dayA1 to (3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20) with context = cycle27

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle27

Setting value of dayC2 to (3 4 5) with context = cycle27

Setting value of dayB1 to (3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle27

Setting value of dayA4 to (3 4 5) with context = cycle27

USING CONSTRAINT: C6_one

Setting value of dayA2 to (8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25) with context = cycle27

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayC2 to (5) with context = cycle28

Setting the value of dayC2 to (4) with context = cycle29

Setting the value of dayC2 to (3) with context = cycle30

USING CONSTRAINT: C8_one

Setting value of dayA3 to (13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle30

USING CONSTRAINT: c1_aux

Setting value of dayC3 to (4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle30

Setting value of dayC1 to (4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle30

USING CONSTRAINT: C2_one

Setting value of dayB1 to (4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle30

Setting value of dayA4 to (4 5) with context = cycle30

USING CONSTRAINT: C3_aux

Setting value of dayC3 to (5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle30

Setting value of dayC1 to (5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle30

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayA4 to (5) with context = cycle31

Setting the value of dayA4 to (4) with context = cycle32

USING CONSTRAINT: c1_aux

Setting value of dayA1 to (3 5 7 8 9 10 11 12 13 15 16 17 18 19 20) with context = cycle32

USING CONSTRAINT: C2_one

Setting value of dayB1 to (5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle32

USING CONSTRAINT: C3_aux

Setting value of dayA1 to (3 7 8 9 10 11 12 13 15 16 17 18 19 20) with context = cycle32

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayA3 to (30) with context = cycle33

Setting the value of dayA3 to (29) with context = cycle34

Setting the value of dayA3 to (25) with context = cycle35

Setting the value of dayA3 to (24) with context = cycle36

Setting the value of dayA3 to (23) with context = cycle37

Setting the value of dayA3 to (21) with context = cycle38

Setting the value of dayA3 to (20) with context = cycle39

Setting the value of dayA3 to (19) with context = cycle40

Setting the value of dayA3 to (18) with context = cycle41

Setting the value of dayA3 to (17) with context = cycle42

Setting the value of dayA3 to (16) with context = cycle43

Setting the value of dayA3 to (15) with context = cycle44

Setting the value of dayA3 to (13) with context = cycle45

USING CONSTRAINT: C6_one

Setting value of dayA2 to (8) with context = cycle45

Setting value of dayA1 to (3) with context = cycle45

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle45

USING CONSTRAINT: C2_two

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 9 10 11 12 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle45

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (5 7 8 9 10 11 12 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle45

Setting value of dayB1 to (5 7 8 9 10 11 12 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle45

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayC1 to (30) with context = cycle46

Setting the value of dayC1 to (29) with context = cycle47

Setting the value of dayC1 to (25) with context = cycle48

Setting the value of dayC1 to (24) with context = cycle49

Setting the value of dayC1 to (23) with context = cycle50

```
Setting the value of dayC1 to (21) with context = cycle51

Setting the value of dayC1 to (20) with context = cycle52

Setting the value of dayC1 to (19) with context = cycle53

Setting the value of dayC1 to (18) with context = cycle54

Setting the value of dayC1 to (17) with context = cycle55

Setting the value of dayC1 to (16) with context = cycle56

Setting the value of dayC1 to (15) with context = cycle57

Setting the value of dayC1 to (12) with context = cycle58

Setting the value of dayC1 to (11) with context = cycle59

Setting the value of dayC1 to (10) with context = cycle60

Setting the value of dayC1 to (9) with context = cycle61

Setting the value of dayC1 to (7) with context = cycle62

Setting the value of dayC1 to (5) with context = cycle63

USING CONSTRAINT: c1_aux

 Setting value of dayC3 to (7 8 9 10 11 12 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle63

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle63
```

Here is the first BACKTRACKING point. It found a problem at constraint C7_aux. It does a chronological BACKTRACKING by just purging the context cycle63.

We shall break the program to look at the current state of the problem. It would be interesting to find out the cause of the failure.

```
USING CONSTRAINT: c1_aux

 Setting value of dayC3 to (5 8 9 10 11 12 15 16 17 18 19 20 21 23 24 25 29 30) with context = cycle62

USING CONSTRAINT: C7_aux


BACKTRACKING... purging context cycle62

[load constraints.l]
USING CONSTRAINT: C8

^C
Interrupt:
Break nil
<1>: (describe 'bat_A)
actA4 = (act_B)    cycle = cycle
rangeA4 = (range_C)    cycle = cycle
dayA4 = (4)    cycle = cycle32
```

```
actA3 = (act_A)   cycle = cycle
rangeA3 = (range_A)   cycle = cycle9
dayA3 = (13)   cycle = cycle45
actA2 = (act_A)   cycle = cycle
rangeA2 = (range_A)   cycle = cycle11
dayA2 = (8)   cycle = cycle45
actA1 = (act_A)   cycle = cycle
rangeA1 = (range_A)   cycle = cycle13
dayA1 = (3)   cycle = cycle45
nil
<1>:

(describe 'batB_B)
actB3 = (act_C)   cycle = cycle
rangeB3 = (range_B)   cycle = cycle
dayB3 = (26)   cycle = cycle23
actB2 = (act_A)   cycle = cycle
rangeB2 = (range_A)   cycle = cycle7
dayB2 = (27)   cycle = cycle20
actB1 = (act_B)   cycle = cycle
rangeB1 = (range_C)   cycle = cycle
dayB1 = (5 7 8 9 10 11 12 15 16 17 18 19 20 21 23 24 25 29 30)   cycle = cycle45
nil
<1>:

(describe 'bat_C)
actC3 = (act_B)   cycle = cycle
rangeC3 = (range_C)   cycle = cycle
dayC3 = (5 7 8 9 10 11 12 15 16 17 18 19 20 21 23 24 25 29 30)   cycle = cycle45
actC2 = (act_B)   cycle = cycle
rangeC2 = (range_C)   cycle = cycle
dayC2 = (3)   cycle = cycle30
actC1 = (act_A)   cycle = cycle
rangeC1 = (range_A)   cycle = cycle5
dayC1 = (9)   cycle = cycle61
nil
<1>:

(describe 'bat_D)
actD2 = (act_A)   cycle = cycle
rangeD2 = (range_A)   cycle = cycle1
dayD2 = (26)   cycle = cycle17
actD1 = (act_A)   cycle = cycle
rangeD1 = (range_A)   cycle = cycle3
dayD1 = (2)   cycle = cycle27
nil
```

We see that the backup was initiated by using the constraint C7_aux. This constraint says that if activity act_A is scheduled on range_A on any day 'x'. Then range_C will be unsafe to use on that day.

We see that

rangeA1 = range_A

actA1 = act_A

dayA1 = (3) Cycle = 45

rangeC2 = range_C

dayC2 = (3) Cycle = 30

With the current context, however, is cycle61. The stupid program does not realize that it should backtrack to cycle45 to get rid of the problem. It will backtrack and try to propagate 20 times. I hereby force the backup just to save time. This is the problem in chronological BACKTRACKING.

I take the liberty of backing up to cycle45.... This is done despite the fact the program will (eventually) reach do the same.

```
<1>: (loop for x from 45 to 61 do (backtrack))

BACKTRACKING... purging context cycle61

BACKTRACKING... purging context cycle60

BACKTRACKING... purging context cycle59

BACKTRACKING... purging context cycle58

BACKTRACKING... purging context cycle57

BACKTRACKING... purging context cycle56

BACKTRACKING... purging context cycle55

BACKTRACKING... purging context cycle54

BACKTRACKING... purging context cycle53

BACKTRACKING... purging context cycle52

BACKTRACKING... purging context cycle51

BACKTRACKING... purging context cycle50

BACKTRACKING... purging context cycle49
```

```
BACKTRACKING... purging context cycle48


BACKTRACKING... purging context cycle47


BACKTRACKING... purging context cycle46


BACKTRACKING... purging context cycle45

nil
<1>: (reset)

[Return to top level]
-> (describe 'bat_A)
actA4 = (act_B)    cycle = cycle
rangeA4 = (range_C)    cycle = cycle
dayA4 = (4)    cycle = cycle32
actA3 = (act_A)    cycle = cycle
rangeA3 = (range_A)    cycle = cycle9
dayA3 = (15)    cycle = cycle44
actA2 = (act_A)    cycle = cycle
rangeA2 = (range_A)    cycle = cycle11
dayA2 = (8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25)    cycle = cycle27
actA1 = (act_A)    cycle = cycle
rangeA1 = (range_A)    cycle = cycle13
dayA1 = (3 7 8 9 10 11 12 13 15 16 17 18 19 20)    cycle = cycle32
nil
->

(describe 'bat_B)
actB3 = (act_C)    cycle = cycle
rangeB3 = (range_B)    cycle = cycle
dayB3 = (26)    cycle = cycle23
actB2 = (act_A)    cycle = cycle
rangeB2 = (range_A)    cycle = cycle7
dayB2 = (27)    cycle = cycle20
actB1 = (act_B)    cycle = cycle
rangeB1 = (range_C)    cycle = cycle
dayB1 = (5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30)    cycle = cycle32
nil
-> (describe 'bat_C)
actC3 = (act_B)    cycle = cycle
rangeC3 = (range_C)    cycle = cycle
dayC3 = (5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30)    cycle = cycle30
actC2 = (act_B)    cycle = cycle
rangeC2 = (range_C)    cycle = cycle
dayC2 = (3)    cycle = cycle30
actC1 = (act_A)    cycle = cycle
rangeC1 = (range_A)    cycle = cycle5
dayC1 = (5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 29 30)    cycle = cycle30
nil
->

(describe 'bat_D)
actD2 = (act_A)    cycle = cycle
rangeD2 = (range_A)    cycle = cycle1
dayD2 = (26)    cycle = cycle17
actD1 = (act_A)    cycle = cycle
rangeD1 = (range_A)    cycle = cycle3
dayD1 = (2)    cycle = cycle27
```

nil

```
-> (search)

USING CONSTRAINT: C6_one

 Setting value of dayA2 to (8 9 10) with context = cycle44

 Setting value of dayA1 to (3) with context = cycle44

USING CONSTRAINT: C2_one

 Setting value of dayC1 to (5 7 8 9 10 11 12 13 16 17 18 19 20 21 23 24 25 29 30) with context = cycle44

USING CONSTRAINT: C7_aux

 Setting value of dayC3 to (5 7 8 9 10 11 12 13 16 17 18 19 20 21 23 24 25 29 30) with context = cycle44

 Setting value of dayB1 to (5 7 8 9 10 11 12 13 16 17 18 19 20 21 23 24 25 29 30) with context = cycle44

PROPAGATION COMPLETE

 BRANCHING ...

Setting the value of dayA2 to (10) with context = cycle45

Setting the value of dayA2 to (9) with context = cycle46

Setting the value of dayA2 to (8) with context = cycle47

USING CONSTRAINT: C2_one

 Setting value of dayC1 to (5 7 9 10 11 12 13 16 17 18 19 20 21 23 24 25 29 30) with context = cycle47

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle47

USING CONSTRAINT: C2_one

 Setting value of dayC1 to (5 7 8 10 11 12 13 16 17 18 19 20 21 23 24 25 29 30) with context = cycle46

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle46

USING CONSTRAINT: C2_one

 Setting value of dayC1 to (5 7 8 9 11 12 13 16 17 18 19 20 21 23 24 25 29 30) with context = cycle45

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle45
```

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle44

USING CONSTRAINT: C6_one

Setting value of dayA2 to (8 9 10 11) with context = cycle43

Setting value of dayA1 to (3) with context = cycle43

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 8 9 10 11 12 13 15 17 18 19 20 21 23 24 25 29 30) with context = cycle43

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (5 7 8 9 10 11 12 13 15 17 18 19 20 21 23 24 25 29 30) with context = cycle43

Setting value of dayB1 to (5 7 8 9 10 11 12 13 15 17 18 19 20 21 23 24 25 29 30) with context = cycle43

USING CONSTRAINT: C1

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayA2 to (11) with context = cycle44

Setting the value of dayA2 to (10) with context = cycle45

Setting the value of dayA2 to (9) with context = cycle46

Setting the value of dayA2 to (8) with context = cycle47

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 9 10 11 12 13 15 17 18 19 20 21 23 24 25 29 30) with context = cycle47

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle47

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 8 10 11 12 13 15 17 18 19 20 21 23 24 25 29 30) with context = cycle46

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle46

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 8 9 11 12 13 15 17 18 19 20 21 23 24 25 29 30) with context = cycle45

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle45

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 8 9 10 12 13 15 17 18 19 20 21 23 24 25 29 30) with context = cycle44

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle44

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle43

USING CONSTRAINT: C6_one

Setting value of dayA2 to (8 9 10 11 12) with context = cycle42

Setting value of dayA1 to (3 7) with context = cycle42

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 7 8 9 10 11 12 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle42

USING CONSTRAINT: C2_two

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (5 7 8 9 10 11 12 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle42

Setting value of dayB1 to (5 7 8 9 10 11 12 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle42

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayA1 to (7) with context = cycle43

Setting the value of dayA1 to (3) with context = cycle44

USING CONSTRAINT: C7_aux

BACKTRACKING... purging context cycle44

USING CONSTRAINT: C6_one

Setting value of dayA2 to (12) with context = cycle43

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 8 9 10 11 12 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle43

USING CONSTRAINT: C2_one

Setting value of dayC1 to (5 8 9 10 11 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle43

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (5 8 9 10 11 12 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle43

Setting value of dayB1 to (5 8 9 10 11 12 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle43

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayC1 to (30) with context = cycle44

Setting the value of dayC1 to (29) with context = cycle45

Setting the value of dayC1 to (25) with context = cycle46

Setting the value of dayC1 to (24) with context = cycle47

Setting the value of dayC1 to (23) with context = cycle48

Setting the value of dayC1 to (21) with context = cycle49

Setting the value of dayC1 to (20) with context = cycle50

Setting the value of dayC1 to (19) with context = cycle51

Setting the value of dayC1 to (18) with context = cycle52

Setting the value of dayC1 to (16) with context = cycle53

Setting the value of dayC1 to (15) with context = cycle54

Setting the value of dayC1 to (13) with context = cycle55

Setting the value of dayC1 to (11) with context = cycle56

Setting the value of dayC1 to (10) with context = cycle57

Setting the value of dayC1 to (9) with context = cycle58

Setting the value of dayC1 to (8) with context = cycle59

Setting the value of dayC1 to (5) with context = cycle60

USING CONSTRAINT: c1_aux

Setting value of dayC3 to (8 9 10 11 12 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle60

USING CONSTRAINT: C7_aux

Setting value of dayC3 to (8 9 10 11 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle60

Setting value of dayB1 to (5 8 9 10 11 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle60

USING CONSTRAINT: C7_aux

Setting value of dayB1 to (8 9 10 11 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle60

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayC3 to (30) with context = cycle61

Setting the value of dayC3 to (29) with context = cycle62

Setting the value of dayC3 to (25) with context = cycle63

Setting the value of dayC3 to (24) with context = cycle64

Setting the value of dayC3 to (23) with context = cycle65

Setting the value of dayC3 to (21) with context = cycle66

Setting the value of dayC3 to (20) with context = cycle67

Setting the value of dayC3 to (19) with context = cycle68

Setting the value of dayC3 to (18) with context = cycle69

Setting the value of dayC3 to (16) with context = cycle70

Setting the value of dayC3 to (15) with context = cycle71

Setting the value of dayC3 to (13) with context = cycle72

Setting the value of dayC3 to (11) with context = cycle73

Setting the value of dayC3 to (10) with context = cycle74

Setting the value of dayC3 to (9) with context = cycle75

Setting the value of dayC3 to (8) with context = cycle76

USING CONSTRAINT: C2_one

Setting value of dayB1 to (9 10 11 13 15 16 18 19 20 21 23 24 25 29 30) with context = cycle76

PROPAGATION COMPLETE

BRANCHING ...

Setting the value of dayB1 to (30) with context = cycle77

Setting the value of dayB1 to (29) with context = cycle78

Setting the value of dayB1 to (25) with context = cycle79

Setting the value of dayB1 to (24) with context = cycle80

```
Setting the value of dayB1 to (23) with context = cycle81

Setting the value of dayB1 to (21) with context = cycle82

Setting the value of dayB1 to (20) with context = cycle83

Setting the value of dayB1 to (19) with context = cycle84

Setting the value of dayB1 to (18) with context = cycle85

Setting the value of dayB1 to (16) with context = cycle86

Setting the value of dayB1 to (15) with context = cycle87

Setting the value of dayB1 to (13) with context = cycle88

Setting the value of dayB1 to (11) with context = cycle89

Setting the value of dayB1 to (10) with context = cycle90

Setting the value of dayB1 to (9) with context = cycle91

PROPAGATION COMPLETE

BRANCHING ...

SCHEDULE GENERATED SUCCESSFULLY !
```

The program has successfully terminated. We now look at the results. The four battalions are described below.

At the end, we look at the stack of one of the variables. It is interesting to note how the value of dayA2 changed from one context to another. Note that the cycle numbers jump from nil to 27 to 42.

```
-> (describe 'bat_A)
actA4 = (act_B)    cycle = cycle
rangeA4 = (range_C)    cycle = cycle
dayA4 = (4)    cycle = cycle32
actA3 = (act_A)    cycle = cycle
rangeA3 = (range_A)    cycle = cycle9
dayA3 = (17)    cycle = cycle42
actA2 = (act_A)    cycle = cycle
rangeA2 = (range_A)    cycle = cycle11
dayA2 = (12)    cycle = cycle43
actA1 = (act_A)    cycle = cycle
rangeA1 = (range_A)    cycle = cycle13
dayA1 = (7)    cycle = cycle43
nil
-> (describe 'bat_B)
actB3 = (act_C)    cycle = cycle
rangeB3 = (range_B)    cycle = cycle
dayB3 = (26)    cycle = cycle23
actB2 = (act_A)    cycle = cycle
rangeB2 = (range_A)    cycle = cycle7
dayB2 = (27)    cycle = cycle20
actB1 = (act_B)    cycle = cycle
rangeB1 = (range_C)    cycle = cycle
dayB1 = (9)    cycle = cycle91
nil
```

```
-> (describe 'bat_C)
actC3 = (act_B)    cycle = cycle
rangeC3 = (range_C)    cycle = cycle
dayC3 = (8)    cycle = cycle76
actC2 = (act_B)    cycle = cycle
rangeC2 = (range_C)    cycle = cycle
dayC2 = (3)    cycle = cycle30
actC1 = (act_A)    cycle = cycle
rangeC1 = (range_A)    cycle = cycle5
dayC1 = (5)    cycle = cycle60
nil
->

(describe 'bat_D)
actD2 = (act_A)    cycle = cycle
rangeD2 = (range_A)    cycle = cycle1
dayD2 = (26)    cycle = cycle17
actD1 = (act_A)    cycle = cycle
rangeD1 = (range_A)    cycle = cycle3
dayD1 = (2)    cycle = cycle27
nil
-> (get 'dayD1 'values                ()
nil
t


-> (apply 'pp (get 'dayA2 'values))
(cycle43 (12))
(cycle42 (8 9 10 11 12))
(cycle27 (8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25))
(cycle (7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25))
(cycle (2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30))
(cycle (1 2 3 4 5 7 8 9 10 11 12 13 15 16 17 18 19 20 21 23 24 25 26 27 29 30))
cycle91
```

hera% ^D script done on Wed May 8 12:03:22 1985

DISTRIBUTION

Chief of Engineers

ATTN: DAEN-RDM
ATTN: DAEN-ZCE
ATTN: DAEN-ZCF
ATTN: DAEN-ZCI
ATTN: DAEN-ZCM

FESA, ATTN: Library 22060

US Army Engineer Districts
ATTN: Library (41)

US Army Engineer Divisions
ATTN: Library (14)

CRREL, ATTN: Library 03755

WES, ATTN: Library 39180

NCEL, ATTN: Library, Code L08A 93041

Defense Technical Info. Center 22314
ATTN: DDA (2)

US Govt Printing Office 22304
Receiving Sect/Depository Copies (2)

Army Development and Employment Agency (ADEA)
ATTN: MODE-FDD-TDB
Ft. Lewis, WA 98433-5000

ADEA
P.O. Box 33368
Ft. Lewis, WA 98433-0368

Commander
Army Training Board
ATTN: ATTG-BT
Ft. Monroe, VA 23651

The Army Library (ANRAL-R) 20310
ATTN: Army Studies Section

ATE
LMED
-8