CONSTRAINT PROGRAMMING APPROACH TO THE TAEMS SCHEDULING PROBLEM

By

Soumita Datta

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2006

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor Sandeep Neema

To my parents.

ACKNOWLEDGEMENTS

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA), Information Technology Office, Coordinators program.

Firstly, I would like to take this opportunity to express my sincere gratitude towards my academic and research advisor Dr. Gabor Karsai for helping me through my academic pursuit. I am grateful to him for introducing me to the field of constraint programming techniques, for motivating me for this research and for his patience and kindness. I also wish to thank Dr. Sandeep Neema, my second approver, for his invaluable insights.

I would like to thank Chris vanBuskirk for patiently answering my innumerable questions and helping me to understand the problem better. My thanks to Will Vaughan and Sameer Singh as well for their kind help in understanding nuances of TAEMS. Many thanks to Himanshu Neema and other members of the ISIS ANTS team for their support.

Finally, I would like to thank my parents for always motivating me, my friends for always cheering me, and my husband for always being there for me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI – Artificial Intelligence.

TAEMS – Task Analysis, Environment Modeling, and Simulation

DTC – Design-to-Criteria

API – Application Programming Interface

CSP – Constraint Satisfaction Problem

CLP – Constraint Logic Programming

HTN – Hierarchical Task Network

DAG – Directed Acyclic Graph

QAF – Quality Accumulation Function

NLE – Non-Local Effect

CP – Constraint Programming

IC – Interval Constraint

RIA – Real Number Interval Arithmetic

FD – Finite Domain

BBS – Bounded Backtrack Search

LDS – Limited Discrepancy Search

UML – Unified Modeling Language

CoA – Course of Action

XML – Extensible Markup Language

JVM – Java Virtual Machine

CHAPTER I

INTRODUCTION

Planning is deciding upon a course of action before acting
– By Patrick Doyle

Planning and Scheduling is a broad term used in the field of AI to address goal directed problem solving. Planning [8] involves reasoning about future actions to sequence and generate a reasonable series of actions to be taken in order to achieve a goal. The planning issue is common to Artificial Intelligence methods where an agent has to come up with an action sequence that leads from an initial state to a goal state. In general, the term planning is used to describe a wide variety of problems. Scheduling [1] on the other hand, deals with temporal constraints on the tasks and resource assignments to the task. For example, a task can have predecessor tasks, which must be completed before it can start. Many tasks can share a common set of resources etc. The scheduler output can be a sequenced set of tasks with their start times assigned in such a way that there is no conflict during task execution.

Introduction To Planning And Scheduling

As mentioned earlier, planning can be thought of as determining and sequencing all the small tasks that must be carried out in order to accomplish a bigger goal. A common example used in this context is the plan to buy milk from local store [26]. It may sound like a simple task, but if it is broken down, there are many small tasks involved like get keys, get purse, get into car, drive to local store, get milk, buy milk, etc. Planning also considers several constraints, which

control when certain tasks can or cannot happen. Two of the many constraints in the previous example are, one must get keys and purse before driving to the store and one must get the milk before buying it.

A simple plan for buying milk at the store with above assumptions is shown in Figure 1:

Figure 1. A Simple Plan to Buy Milk from Local Store

After a plan is obtained, the process of determining whether enough resources are available to carry out the plan and whether other hard temporal and timeline constraints are met is called scheduling. Revisiting the milk-buying example, two resources that scheduling would have to take into account are fuel and time. If two gallons of gas is required to get to the store and back and the car only has one gallon, a plan has to be developed which includes a stop at the gas station. Besides, if it takes 15 minutes to drive to the store, the store closes at 10:00, and it is currently 9:30, one must also take that time constraint into account when scheduling tasks. All the above constraints lead to a schedule which has the plan of sequenced tasks along with associated start time and resource allocation. The coordination problem described in the following subsection deals with planning and scheduling in a multi-agent environment where more than one agent are required to coordinate their plans and schedules in order to achieve a goal.

Coordination Problem And The TAEMS Framework

The Agent Coordination Problem [14] is a well-known problem in AI. For any practical distributed system, the inextricable relationship between the domain and its components leads to the coordination problem. One such coordination problem is the distributed schedule coordination involving multiple agents [7]. The solution to such a problem involves computing an execution plan and schedule for the tasks assigned to various agents often under uncertain conditions. In AI terminology, an agent is considered to be an entity that has some knowledge or belief about the environment and can perform some actions on the environment. The agent may have choice of actions that it can perform on the environment to accomplish a particular task. A particular ordering or timing of these actions may affect the performance of the agent. This gives rise to the basic coordination problem which involves choosing and temporally ordering the actions (many of which might be interrelated) such that the overall task is accomplished by the agent. The problem is worsened by the complexity and uncertainty in the task structure. An agent might have only a partial view of the task structure. Even if it has the full view, the structure might change dynamically while the agent is not sure about the result of its own actions. Many such agents are present in a realistic environment. In such a multi-agent environment [14], coordination becomes even more complex, where agents can affect each others actions in a helpful or adverse manner. Each of these agents might have their own incomplete version of the overall task structure. The planning problem [8] for such a situation involves choosing a set of actions which if performed would accomplish a goal and the scheduling problem involves choosing execution start times for these actions so that the actions could be assigned to the agents. If a goal is shared by more than one agent, multiple plans might be possible. The idea is

to identify possible conflicts in such plans and generate one which would avoid adversarial behavior.

An effective framework to depict such agent coordination problems in computational environments is TAEMS [9] the acronym for **T**ask **A**nalysis, **E**nvironment **M**odeling, and **S**imulation. It provides a systematic method to formally represent agent coordination problems in a domain-independent manner. A TAEMS model representation of a problem highlights the basic coordination problem and phases out the domain details. The basic coordination problem refers to finding a temporal ordering of all possible actions to accomplish a Task. Most of the computational environments can be specified, analyzed, and simulated using the TAEMS framework. Though the TAEMS representation itself does not represent a plan or schedule, it describes the task structure of the agents and any interdependencies between them. A task structure as defined in [9] is the representation of the agent capabilities or the tasks the agent may perform. TAEMS also provides means for simulating environments, generating random episodes, providing subjective information to the agents, and tracking their performance. A simple TAEMS task structure is shown in Figure 2. A Simple TAEMS Task Structure

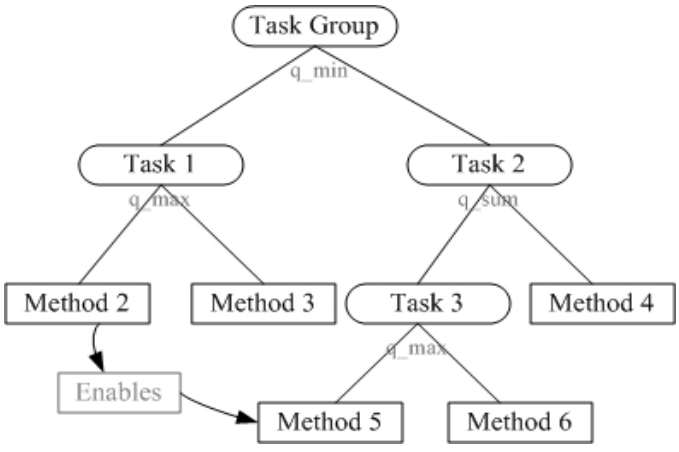


Figure 2. A Simple TAEMS Task Structure

The TAEMS task structure typically looks like a tree structure with interdependencies. These interdependencies actually make the representation a directed acyclic graph rather than a tree. The graph node with no incoming arc or parent node is the root task of the task structure, representing the high-level job that needs to be done. It is an abstraction of the actual action to be performed. The directed arcs from this parent node go to its sub-tasks, at least one or at most all of which can be performed to achieve the parent task in some way. This task-subtask structure continues till it is grounded by the 'methods' which are the actual low level actions that the agent performs to achieve the goal at root task. When an agent executes a primitive action, some performance criteria are associated with such an execution. These criteria signify the degree of success of the agent in performing the action. The main performance criterion associated with a TAEMS structure is quality. Quality is a rather abstract term in this context and it assumes relevance depending on the particular environment. Examples of quality measures include the precision, belief, or completeness of a method execution. Such an execution accrues some quality and this quality, in turn, is accumulated by its parent tasks and propagated to the root task. The ultimate goal is to have some quality at the root task. The plan consists of an ordering of these methods, and the schedule consists of assigning start times to these methods which if executed within their expected duration would achieve some quality at the root task. There may be multiple plans and corresponding schedules for task structures each of which may accomplish the root task if performed as per the schedule and within the expected duration. The problem is complicated by the presence of interrelations between tasks and methods which makes this scheduling problem an NP-hard problem [27]. Various partial schedulers available to solve the TAEMS scheduling problem are a) DTC - Design-to-Criteria scheduler [15] b) BIG - Bounded Information Gathering [28] c) IHome - The Intelligent Home Project [17].

In this thesis, a constraint programming approach is utilized to solve the TAEMS scheduling problem.

Introduction to Constraint Programming

Constraint programming [2] is a problem solving paradigm which establishes the distinction between precise declarative description of constraints that define the problem and the search algorithms and heuristics that enable selection and cancellation of decisions to solve the problem [9]. Constraint Programming is particularly useful in solving difficult combinatorial problems which require searching through a possibly exponential sized solution space in order to get a solution. The idea is that instead of implementing hand-crafted algorithms to generate initial schedules in the application, a constraint programming framework and its optimized built-in algorithms can be used to achieve the same objective. More details on Constraint Programming have been provided in the background section (Chapter II).

Problem Description

This thesis aims to generate the initial schedule for a TAEMS style objective task structure using constraint programming techniques. The scope of this thesis is limited to generating the initial plan and schedule for the task structure under the assumption that the expected start times and duration of methods involved are a good approximation of the actual start time and duration. It does not take into consideration the probabilistic nature of the method outcomes or attempts at agent negotiation, plan repair or dynamic behavior of the system. It also does not attempt to optimize the root task quality, which is beyond the scope of this work. Solving this initial planning and scheduling problem using constraint programming techniques involves encoding the TAEMS problem as a CSP (Constraint Satisfaction Problem) using

various solver search techniques and decoding the solution back to a TAEMS plan and schedule. The solver of choice used here is ECLiPSe [6] (a Prolog variant) version 5.3 and encoding and decoding has been accomplished using Java, making use of best of both worlds. ECLiPSe is simple to use and enables fast encoding of the problem and has several search algorithms. It provides excellent support for embedding the solver in other high level languages like C, C++, Java, Tcl etc. The Java front-end encoder-decoder together with an ECLiPSe back-end Solver provides an easy, versatile framework for solving the TAEMS scheduling problem.

This thesis is organized in the following order. Chapter 2 provides an introduction to the TAEMS framework and its details. It also provides an introduction to constraint programming and constraint programming techniques along with planning, scheduling problems and encoding. The technical details of the encoding, decoding and search control techniques used to solve the problem are provided in Chapter 3. Terms used in context of the thesis are also defined in this chapter. Chapter 4 shows the result of the encoding technique as applied to a small, medium and large scale TAEMS problem and analysis of the result. Finally, the thesis is concluded in Chapter 5 along with some pertinent remarks regarding the future work relating to this thesis.

CHAPTER II

BACKGROUNDS

This chapter gives a preview to the various concepts and terminologies used in this thesis. There are many approaches to the general planning and scheduling problem. These are briefly discussed in this chapter. A constraint programming approach has been taken here to generate schedules for TAEMS style task structures. To elucidate the approach taken, this chapter focuses on the basic concepts of the TAEMS, semantics of a TAEMS model, Constraint Programming and Solver Technology.

Planning and Scheduling

The introduction section presented a brief overview of Planning and Scheduling concepts. Here the problem is defined more formally and the nuances of the area are detailed. In AI, planning is used to denote coming up with a sequence of formally-described world-states. A planning domain [8] contains a set of operators or action types. Each operator may be executed only in some particular set of world states which are preconditions, and has some particular set of effects on its world state (its effects). A planning domain with an initial world state, and a desired goal state (or set of goal states) of the world makes up a planning problem. A planner solves a planning problem by producing a sequence of actions, each of which is legal in its starting world state, which takes the initial state to a goal state. A common example of planning domain is the blocks world, which consist of a model of stacks of blocks on an infinite table [26].

Once a plan is obtained it needs to be scheduled. Scheduling [1] is the problem of assigning a set of tasks to a set of resources subject to a set of constraints. Examples of

scheduling constraints include deadlines (e.g., job1 must be completed by time t), resource capacities (e.g., there are only four machines), precedence constraints on the order of tasks (e.g., a piece must be sanded before it is painted), and priorities on tasks (e.g., finish job j as soon as possible while meeting the other deadlines). Examples of scheduling domains include classical job-shop scheduling, manufacturing scheduling, and transportation scheduling.

Planning and scheduling can be distinguished by the fact that scheduling is mainly concerned with finding *when* to carry out actions while planning is concerned with *what* actions need to be performed. In practice this difference often is not clear and many real-world problems involve figuring out both what and when.

Planning Approaches

There are several approaches to the planning problem described above. All these approaches have some basic assumptions. Firstly, it is assumed that the agent's action changes the world from one static state to another. Also, effects of actions are predictable. The world changes only as the result of the agent's actions. Few basic approaches to planning [8] are described below:

1. Non-hierarchical – This kind of planning consists of finding a sequence of operators to achieve each of the goals. This planner does not differentiate between important goals and less critical ones, and so it can waste considerable amounts of time finding solutions to non-critical parts of a plan, only to find it can't solve a critical part. It is a problem with this particular approach. Examples of this type of planner include STRIPS, HACKER, and WARPLAN [8].
2. Hierarchical – This type of planning aims to compute a plan that describes how to take actions in levels of increasing refinement and specificity [26]. Most plans are hierarchical in

nature. The main idea here is to simplify the search and reasoning process by finding vague solutions at levels where the details are not computationally overwhelming, and then refine them. Examples of this type of planner include ABSTRIPS [13], NOAH, and SIPE.

3. Conditional Planning – This kind of planning deals with incomplete information about the world by constructing a plan that accounts for each possible contingency that may arise. Examples of such planner are CNLP [11], C-BURIDAN etc.
4. Opportunistic – This is a type of Real-time planning in which operators are invoked when they are useful in the current state. These planners take a bottom-up approach to planning whereas hierarchical planners take a top-down approach. Such opportunistic planning builds islands in plan space and attempts to connect them, while hierarchical planners try to develop an entire plan at once and then refine it. Example of such a planner is OPM [10].

To explain the basic planning process, one of the non-hierarchical planners STRIPS is elaborated here.

STRIPS Planner

STRIPS is a non-hierarchical planning system. It makes no distinction between more important and less important parts of the plan, so it could be bogged down dealing with unimportant details. Nevertheless, it was one of the first planners to be designed.

STRIPS represents the world as a set of formulae in first-order logic [26]. Each state in the search space consists of a world model and set of goals to be achieved. Goals are maintained in a stack. Whenever the top goal on the stack matches the current state description, it is eliminated, and any match substitution in the goal is propagated down the stack. Otherwise, if the top goal is a compound goal, each component literal is added as a separate goal above it on the stack. This system of planning may suffer from Sussman's Anomaly [26].

Sussman's Anomaly is a famous instance of sub-goal interaction, where the solutions to two separate sub-goals must be interleaved in order to solve them both. No sequential ordering of the sub-goal solutions will work. Specifically, the problem is as follows. In the Blocks World, the initial state consists of C on A on the table and B on the table. The goal state is A on B on C. Planner will break the goal into sub goals A on B and B on C. Once A on B is completed, A is on B and C is on table. To achieve the second goal B on C, B is placed on C and A is on table. This undoes the earlier sub-goal. In STRIPS, if solving one component goal undoes an already solved component of the same compound goal, the undone goal is reconsidered and solved again if needed. Hence it takes care of Sussman's Anomaly.

A STRIPS operator consists of an operator name together with a prerequisite list (or precondition list), an add list, and a delete list. The elements of these lists are all proposition expressions; the precondition list is a conjunction of positive literals which denote the pre-requisite states for the operator, while the add and delete lists are conjunctions of positive or negative literals which add or delete goal states.

If the top goal on the stack is an unsolved single-literal goal, STRIPS finds a production that has that literal in its add list, and replaces the goal with the instantiation of that production, placing the production's preconditions above it on the stack.

Given an initial condition and final state, these STRIPS notation can be used to come up with a sequence of actions to achieve the goal state from the initial state. To explain the concepts discussed here, the kitchen cleaning example is elaborated.

Say, the initial state of the kitchen is: `isDirty(stove) ∧ isClean(sink) ∧ isAbsent(garbage) ∧ isDirty(refrigerator)`. For a completely clean kitchen, the desired goal state is: `IsClean(stove) ∧ IsClean(floor) ∧ IsClean(counters) ∧ IsClean(sink)`

∧ isAbsent(garbage) ∧ IsClean(refrigerator). Now the planning problem is to determine the actions that should be performed in sequence to achieve a clean kitchen (goal state). STRIPS operators can be used to define the actions.

The STRIPS-style operators that might be used for the cleaning actions are as follows:

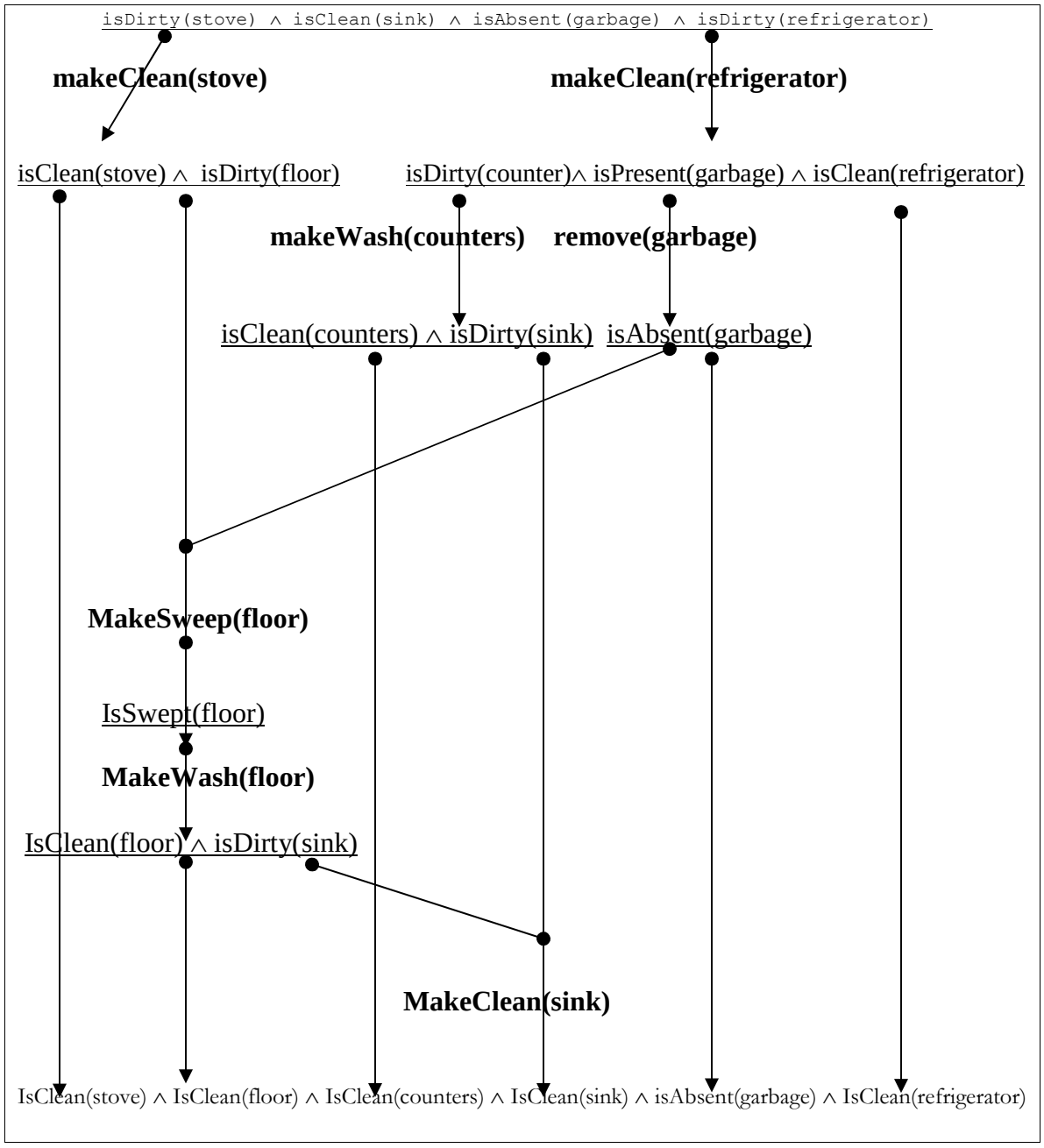
```
Action: clean the stove
makeClean(stove)
Pre-condition:    isDirty(stove)
Delele-list  :    isDirty(stove)
Add-list     :    isClean(stove) ∧ isDirty(floor)

Action: clean the refrigerator
makeClean(refrigerator)
Pre-condition:    isDirty(refrigerator)
Delele-list  :    isDirty(refrigerator)
Add-list     :    isClean(refrigerator) ∧ isDirty(counters) ∧
isPresent(garbage)

Action: Wash the counters
makeWash(counters)
Pre-condition:    isDirty(counters)
Delele-list  :    isDirty(counters)
Add-list     :    isClean(counters) ∧isDirty(sink)

Action: Wash the floor
makeWash(floor)
Pre-condition:    isSwept(floor)
Delele-list  :    isSwept(floor)
Add-list     :    isClean(floor) ∧isDirty(sink)

Action: Sweep the floor
makeSweep(floor)
Pre-condition:    isAbsent(garbage)∧isDirty(floor)
Delele-list  :    isAbsent(garbage)∧isDirty(floor)
Add-list     :    isSwept(floor)

Action: Remove the garbage
remove(garbage)
Pre-condition:    isPresent(garbage)
Delele-list  :    isPresent(garbage)
Add-list     :    isAbsent(garbage)

Action: Clean the sink
makeClean(sink)
Pre-condition:    isDirty(sink)
Delele-list  :    isDirty(sink)
Add-list     :    isClean(sink)
```

Since state isDirty(stove) is at the top of the goal stack (initial state) and pre-condition for makeClean(stove) operator, this operator can be immediately applied. This results in removal of isDirty(stove) from goal stack and addition of isClean(stove) and isDirty(floor) to goal stack. The unresolved literals at goal stack now are: isClean(sink) ∧ isAbsent(garbage) ∧ isDirty(refrigerator) ∧ isClean(stove) ∧ isDirty(floor). In the next step, operator makeClean(refrigerator) can be applied. The unresolved literals at the goal stack now are: isClean(refrigerator) ∧ isDirty(counters) ∧ isClean(sink) ∧ isAbsent(garbage) ∧ isClean(stove) ∧ isDirty(floor). Similarly, the planner decides on the next action depending on the state of the goal stack and operator pre-condition. When the goal stack matches the goal condition the planner stops and prints the complete plan. In the plan path diagram below, goal state is reached from initial state by applying the operators in sequence as shown. In the initial state the only two operators that can be applied are makeClean(stove) and makeClean(refrigerator) in any order. Subsequently the other operators can be applied in sequence. One of the possible plans as obtained from the plan path diagram is:

```
makeClean(stove)->makeClean(refrigerator)->makeWash(counters)->
remove(garbage)->makeSweep(floor)->makeWash(floor)->makeClean(sink).
```

Several such plans can be obtained from the diagram, each of which will lead from the initial state to the goal state successfully. Though this is a trivial example, it explains the basic planning problem and plan generation using STRIPS notation.

The plan path for cleaning a kitchen is depicted in the following diagram.



TAEMS

As mentioned earlier TAEMS [9][25] is a framework for Task **A**nalysis, **E**nvironment **M**odeling, and **S**imulation to represent the agent coordination problems in complex

computational task environments [16] in a formal, domain-independent manner. Example of computational task environments are distributed sensor networks, distributed design problems, complex distributed simulations, and the control processes for almost any distributed or parallel AI application [14]. These environments often require response by the agent within certain deadlines, and the agent might not have all the required information to choose the activities which would eventually lead to optimal performance. The incomplete information that an agent has is modeled in TAEMS as a partial view of the distributed goal tree representation of its activities, which helps prediction about future agent activity and the relative precedence of all activities and their timings. Hence TAEMS helps in modeling the task structure of an agent [12].

A TAEMS task structure typically looks like a complex and/or tree (actually graph) or Hierarchical Task Network (HTN) [9]. It is a hierarchical decomposition framework. To learn more about how TAEMS is used to model a task structure it is worthwhile to know what a task structure is. While designing an intelligent agent system, one might be faced with a situation where the agent has to reason about its potential actions in context of the current surrounding. To reason about such a situation and to act intelligently, the agent has to be answered a lot of queries like what goals should it try to achieve and what should be its course of action to achieve those goals. In short, it should have some representation of what its capabilities are or what tasks it might possibly perform. Task structure is a way to represent this agent capability. Figure 3 shows a TAEMS task structure. This figure is essentially a task decomposition tree. The highest level goal that an agent may try to achieve is represented by the root of this tree. This highest level root node is called the task group. Below this parent node there is a sequence of tasks and methods which illustrate how this task group may be performed.

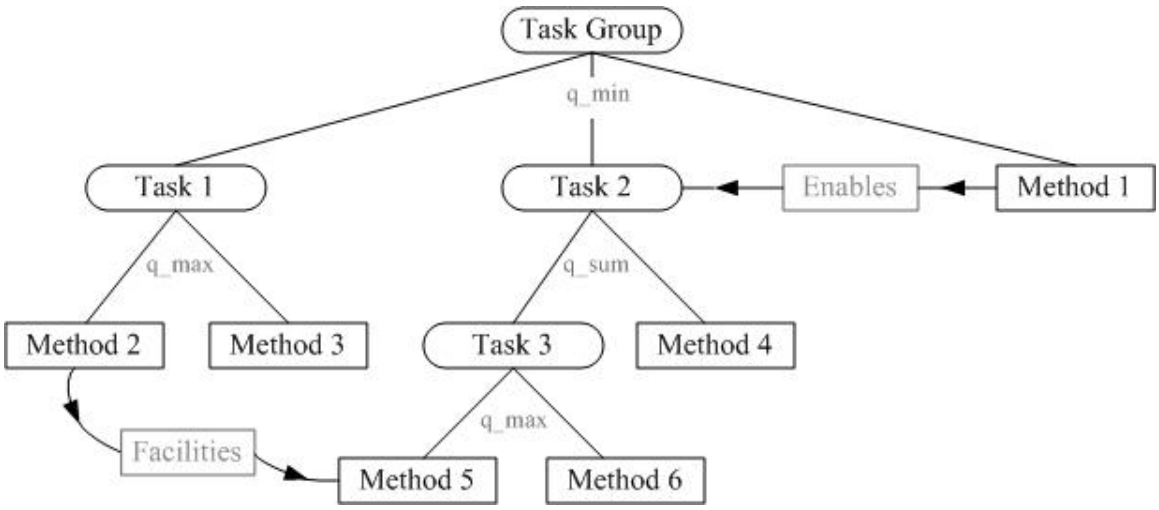


Figure 3. TAEMS Task Structure

Tasks represent sub-goals, which can be further decomposed in the same manner. On the other hand, methods at the leaves of the tree represent executable entities, which are the actual primitive actions the agent will execute to produce some amount of quality. These methods represent the instantiated computations or actions an agent can perform which might eventually achieve the highest level task. Such a method represents a schedulable entity, such as a blackboard knowledge source instance, a chunk of code and its input data etc [16]. A method could also be an instance of a human activity at some useful level of detail, for example, "lift container 1 from position 1". What makes this structure interesting and difficult is the presence of interrelationships between various tasks, sub-tasks and methods. This indicates that execution of some methods will have friendly or adverse effect on the execution of other methods. For example in Figure 3 above, execution of method 2 has friendly effect on execution of method 5 (facilitates relationship). This graphical structure makes TAEMS scheduling an NP-hard problem.

Additionally there are annotations on the task which specify how its sub-tasks may be combined to achieve it. This ably represents the local capabilities of an agent.

Another requirement to model an actual environment is to be able to model potential interactions with other agents. To do this one has to model a series of task structures as the one described above. Tasks may be shared between the trees - indicating that one agent has knowledge of what another agent's capabilities are. Interrelationships may also span the tree which means that methods executed by one agent may affect the other. These interrelationships indicate negotiation or coordination between agents. If one agent can affect another it might be better to intentionally use or avoid those interactions [14].

There is no particular performance criterion in TAEMS. It mainly concentrates on giving two kinds of performance information: the time intervals of task executions, and the quality of the execution. Quality is not a clearly defined term here and assumes relevance depending on the particular environment. Examples of quality measures include the precision, belief, or completeness of a task result. Here the assumption is that quality is a single numeric term with an absolute scale. In real-time problem solving, alternate task execution methods may be available that trade-off time for quality. Hence these are the two performance guidelines that an agent works on.

To go into the details of a task structure described above, its components need to be elaborated. The various building blocks of a task structure are collectively called elements.

The four kinds of TAEMS elements are:

1. Tasks.
2. Methods.
3. Interrelationships
4. Resources

Each of these elements has a labeling name which identifies them in the task structure and an owner agent which possesses the element. Tasks and methods are collectively referred to as nodes [9]. These are basically the elements residing in the graph/tree like portion of the TAEMS task structure. A node that is not a subtask of any task is called a root task. Hence there will be one or more root task in any task structure. When the objective is to determine the schedule for such a structure, one of these root tasks is considered as the task group, whose accumulated quality would actually matter for the problem. Task groups may be thought of as an overall goal associated with a structure, where all the elements suspended beneath it are there to describe how that goal may be achieved. Figure 4 shows a simple structure with a task group (Clean Kitchen), a task and 3 methods (possible nodes).

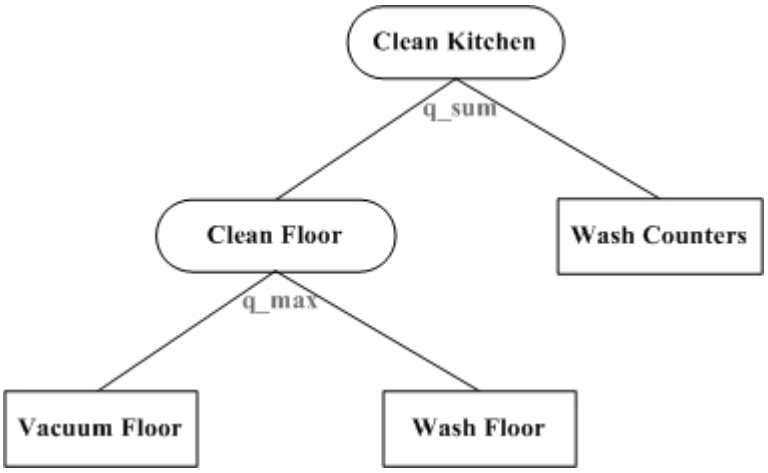


Figure 4. A Simple Task Structure with node types.

To describe a TAEMS structure in a bottom-up fashion, methods need to be mentioned first.

Methods

Methods are the actions that an agent can actually perform while the remainder of the nodes serves to organize the methods in some coherent fashion. The method specification contains information relevant to the execution of the method like the different possible outcomes of the method etc. The method typically has the following mandatory characteristics [24]:

1) A name to identify the method.
2) An agent which is supposed to 'perform' the method or which owns it.
3) Outcomes which is a discrete distribution over possible method outcomes. This is a list like $((o_1,p_1),(o_2,p_2),..,(o_k,p_k))$ where $o_1, o_2, ..., o_k$ are the possible outcomes and $p_1 , p_2 , ..., p_k$ are the probability of these outcomes and $\sum_{i=1}^{k} p_i = 1$. This outcome list describes what the expected quality, duration and cost distributions will be for each possible result the method might achieve when it is executed. Hence, a particular outcome also has a discrete distribution of cost, quality and duration. The duration is expected length of execution of the method. Cost is a unit-less property that defines the "amount" of some tangible or intangible thing that is used when the method is executed. This can be anything depending on the context of the environment which will generally be minimized. Quality is a unit-less relative, abstract characteristics of method execution that the agent would want to maximize. So a higher quality method is better whereas a lower cost method is usually preferred. If each outcome o_i has a quality distribution $((q_{i,1}, p_{i,1}), (q_{i,2}, p_{i,2}), ..., (q_{i,m}, p_{i,m}))$,, the probability that it will execute with quality q is computed as [24]

$$P(q) = \sum_{i,j} p_i * p_{i,j} \ : \ q_{i,j} = q$$

The expected quality $E(q)$ is computed as [24] :

$$E(q) = \sum_{i=1}^{k} \sum_{j=1}^{m} p_i * p_{i,j} * q_{i,j}$$

Duration is computed in exactly the same way.

In the textual representation of TAEMS, known as TTAEMS, a method is textually represented as [9]

```
(spec_method
   (spec_attributes (some_attribute 1))
   (label Method_1)
   (agent Agent_A)
   (supertasks Task_1)
   (outcomes
      (Outcome_1 (density 0.8) (quality_distribution 7.0 1.0)
        (duration_distribution 5.0 0.3 6.0 0.7)(cost_distribution 6.0 1.0)
      )
      (Outcome_2 (density 0.2) (quality_distribution 9.0 1.0)
        (duration_distribution 15.0 0.3 10.0 0.7)(cost_distribution 6.0 1.0)
      )
   )
   (arrival_time 0)
   (earliest_start_time 0)
   (deadline 0)
   (start_time 10)
   (finish_time)
   (accrued_time 3)
   (nonlocal)
)
```

spec_method denotes that the node is a method. The spec_attributes field is sometimes used in a method to store such things as the maximum quality attainable by the method, or indicate its place in the current schedule. Label denotes the unique name of the method which in above case is Method_1. Agent is the agent which owns the method. Supertask is the list of immediate parent tasks of this leaf method. It hints at the fact that the task structure is actually a directed acyclic graph. The outcomes are the possible results of this method execution. Outcome_1 has probability (or density) of 80%, Outcome2 has probability of 20%. Outcome_1 if actually happens will accrue a quality of 7 with 100% probability, can take time of 5 units with probability 30% and 6 units with probability 70%, and will cost 6 units with probability of 100%. Similarly, there is description for Outcome_2. Arrival_time is the actual time that the method

arrives at an agent if this method is part of a negotiation. Earliest_start_time indicates to the agent the earliest possible time at which the method can be executed. For the scheduler this is a hard constraint and limits the method start time. A deadline indicates the latest possible time at which the method should be completed. So if a method starts before its release time or after its deadline it accrues no quality. Actual Start times and finish times will be filled in by the scheduler. The non-local flag indicates whether the method is local to this agent.

Tasks

Methods are in turn organized under tasks. Tasks can also be parent to other tasks leading to a hierarchical structure of tasks and methods. Parent tasks can be accomplished only when one or more of its sub-tasks can be achieved in some way. Tasks are not executable like methods and only represent abstract activities. The main objective of a task is to aggregate its subtasks. This unification of subtasks to accomplish the supertask in some way is referred to as a quality accumulation function, or QAF, which tells how the quality of the subtasks is utilized to compute the quality of the task. More about QAF follows after task discussion. A task will have the following mandatory attributes [24]:

1. A name which identifies the task.
2. An agent who owns the task.
3. Subtasks. Tasks and methods which are child of the current task.
4. Quality Accumulation function.

In the textual representation of TAEMS, called TTAEMS, a task is represented as [9]

```
(spec_task
   (spec_attributes (some_attribute 1))
   (label Task_A)
   (agent Agent_1)
   (supertasks Root)
   (subtasks Task_B Method_A Task_C)
```

```
   (qaf q_min)
   (arrival_time 0)
   (earliest_start_time 0)
   (deadline 10)
)
```

Here spec_task denotes that the node is a task. Supertasks are the parents of the present task whereas subtasks are the child of the present task i.e. those tasks or methods which, when completed or achieved in some manner, may lead to the completion of the parent task. QAF denotes the quality accumulation function. Arrival time, earliest start time and deadline mean the same as method. Earliest start time and deadline impose temporal constraints on the task where a subtask inherits the most restrictive of its own constraints and those of its parent(s). For a node n [24],

$$Earliest-start^{*}(n)=\max(earliest-start(n),\max(earliest-start^{*}(T))),\ T\in supertasks(n)$$

Similarly,

$$deadline^{*}(n)=\min(deadline(n),\min(deadline^{*}(T))),\ T\in supertasks(n)$$

Hence the methods below this task, whose start and finish falls within this earliest start and deadline range will accumulate quality for this task.

Some other facts to ponder about task execution are:

1. A task enters starts state when the first subtask starts after the tasks earliest start time.
2. A task completes when all of its subtasks complete or the task deadline is reached.

A special task called root task is the root node of the task structure and denotes the highest goal an agent is aware of. There may be one or more than one root tasks in a task structure. But the goal that will accumulate quality for the agent for a particular scenario is called task group.

Quality Accumulation Function (QAF)

QAFs are attributes possessed by tasks which are used to compute the quality accumulated by the task based on the quality accumulated by its subtasks. Few QAFs used in TAEMS are explained below.

1) SUM - Example of this QAF is shown in Figure 5, where a simple task structure has a root task and 3 methods. These methods can be activities like, clean kitchen, clean bathroom and clean wardrobe. These methods are essentially unrelated under normal circumstances and more the cleaning the better cleaned the house. In this case a root node of "clean the house" will accumulate a sum of all the method qualities as each method when performed adds to the task group quality. The QAF as described here is called SUM [9]. The sum QAF says that the quality of the super-task is equal to the sum of the qualities of its subtasks irrespective of the order in which methods are actually invoked. This essentially models a process where each additional method the agent chooses to perform will increase the final quality of the task - the accumulated quality increases monotonically as shown in Table 1.

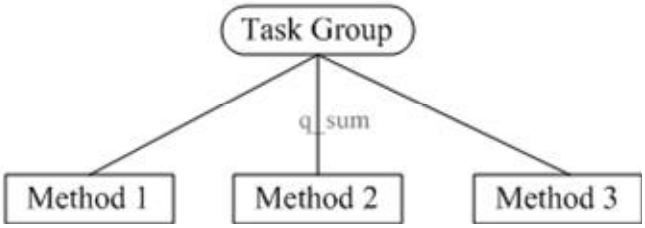


Figure 5. A task group with QAF SUM

Table 1. SUM quality Table

Quality(Action1)	Quality(Action 2)	Quality (Action 3)	Final Quality
Method 1(5)	Method 2(2)	Method 3(7)	14
Method 1(5)	Method 2(2)	-	7
Method 1(5)	Method 3(7)	Method 2(0)	12

2) MAX - Example of this QAF is shown in Figure 6, where a simple task structure has a root task and 3 methods combined using QAF Max. There can be a set of methods like taking a series of tests where the highest score will count towards the grade. At least one of these tests is mandatory for getting a grade. In this case, if grade determines the quality, the maximum of the grades will count towards the final grade. The QAF described here is Max. It is equivalent to an OR operator. It means that the parent task will have the maximum quality of any one of its subtasks as shown in Table 2.

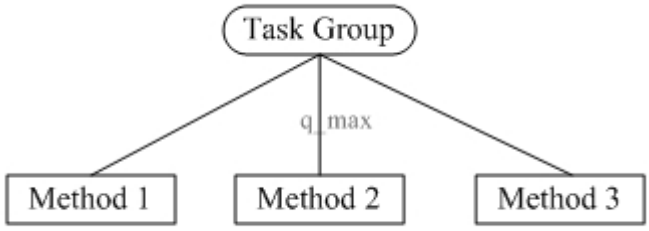


Figure 6. A Task Group with QAF Max

Table 2. Max quality Table

Quality(Action1)	Quality(Action 2)	Quality (Action 3)	Final Quality
Method 1(5)	Method 2(2)	Method 3(7)	7
Method 1(5)	Method 2(2)	-	5
Method 1(5)	Method 3(7)	Method 2(0)	7

3) MIN - Example of this QAF is shown in Figure 7, where a simple task structure has a root task and 3 methods combined using QAF Min. There can be a set of methods where a person is preparing for a party and the methods involve, buying clothes, getting a hair cut and taking a shower. Now if one forgets to do any of these, one will be segregated at the party. Also, if the clothes are not proper or hair cut is odd, there might be unpleasant comments at the party.

So, the reputation at the party will be determined by the worst done task. This is an example of QAF Min.

A Min QAF is functionally equivalent to an AND operator [9]. So the quality of the parent is least among the quality of its subtasks. If a subtask is not performed, then its quality is assumed to be 0, hence to accumulate quality at the root, all the subtasks have to be attempted. This is shown in Table 3.

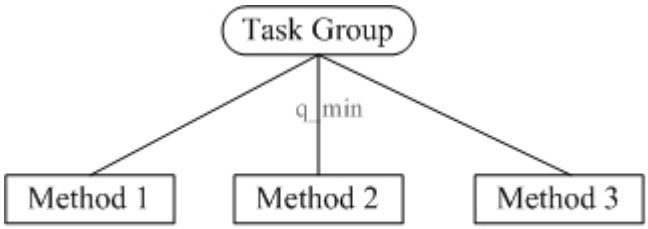


Figure 7. A Task Group with QAF Min

Table 3. Min Quality Table

Quality(Action1)	Quality(Action 2)	Quality (Action 3)	Final Quality
Method 1(5)	Method 2(2)	Method 3(7)	2
Method 1(5)	Method 2(2)	-	0
Method 1(5)	Method 3(7)	Method 2(0)	0

4) SyncSum: Another QAF called syncsum [24] specifies that a task accumulates quality only from subtasks that start at the same time as the task. Hence, this enforces a hard constraint on the task and its subtask start times.

There are other QAFs like All, Exactly_one, Sequential_sum etc that have not been used in the TAEMS model considered here but can be easily incorporated since the encoding concepts are the same.

Interrelationships

A unique feature of TAEMS is its ability to facilitate modeling of the structures linking one node to another (other than task-subtask directed arcs) in the objective task structure which indicates that execution of one node might have an impact on the performance (cost, quality, duration) of the other linked node. These are called Non-local Effects (NLEs) or interrelationships and they span through an agents environment. They have the following attributes:

1. Type – enumerated type which can be {Enables, Disables, Hinders, Facilitates}.
2. Source node – the node of origin of the NLE.
3. Target node – the node of termination of the NLE.
4. Delay – optional delay in source node status change as reflected in target node.

The target node accumulates quality depending on the state of the source node. The source can be a node or a particular outcome of a method. If target node is method then NLE applies to the method. If target node is a task then NLE applies to all the methods below that task hierarchy.

The various kinds of interrelationships are:

1. Enables – Consider the case where there are 2 methods – make tea and drink tea. Now, ‘drink tea’ is only possible after ‘make tea’ (Assuming one drinks only homemade tea). So, one method is pre-requisite of the other. Hence making tea enables drinking tea. This is an example of enables relation between 2 nodes. This type of NLE is of the hard variety and imposes strict ordering conditions on the plan. Formally speaking, the enabled node cannot accumulate quality until the enabling task has accumulated some quality. Hence, the NLE relation is activated when the source gathers some positive quality. Only after the activation

of the NLE, the target node can accumulate quality. Plainly speaking, if some node A enables node B then B cannot be started until A completes.

2. Disables – This NLE is the opposite of enables NLE. It means that if the source node has some positive quality when the disabled node starts then the disabled node cannot accumulate quality. Say, if there are two methods like ‘seal envelope’ and ‘put letter in envelope’. These methods have a precedence relationship in the sense that after ‘seal envelope’ is done there is no way ‘put letter in envelope’ can be achieved. Hence, ‘seal envelope’ disables ‘put letter in envelope’. Plainly speaking, if some node A disables node B then B cannot be achieved if A completes.
3. Facilitates – Other than the hard interrelationships mentioned above there can be soft interrelationships like facilitates. It means that if the source node has some positive quality when the target node starts then the quality of the target node will be increased in some way, whereas the cost and duration will be decreased. There is a particular discrete distribution of this target quality, cost and duration. Example of facilitates NLE would be say two method ‘eat warm leftovers’ and ‘eat cold leftovers’. If there is a task called ‘eating’ , then ‘eat warm leftovers’ will provide more satisfaction to the eating task then ‘eat cold leftovers’. If satisfaction is a measure of quality here then ‘eat warm leftovers’ will facilitate task ‘eating’ by increasing its quality.
4. Hinders – Contrary to Facilitates is Hinders NLE which decreases target quality and increases cost and duration if source node has quality when target node is activated.

One aspect of TAEMS not considered and elaborated in this thesis is the fundamentals of modeling resources (consumable and non-consumable) and the interrelationship involving these (Limits, Consumes, Produces).

TAEMS example – A common and popular example to illustrate the TAEMS concept is the coffee making example from [9].

The model of environmental and task characteristics described above can have two levels: objective and subjective.

The objective view gives the actual view of a particular problem seen from a perspective where an observer knows everything about the problem. It is the true unbiased description of the agent's environment. It has no information about particular agents but just the problem.

The subjective level however describes how agents view and interact with the problem-solving situation. This view is present for each individual agent who must make decisions with only incomplete subjective information. This view is from the perspective of the agent who is supposed to clean the kitchen. It might have no knowledge of a party being planned in the house. So, it does not have a track of all the overall household activities. Coordination is essential in such a situation where agents have to touch base from time to time to make sure their views are not inconsistent from the objective view.

TAEMS Planning and Scheduling

After the discussion above, the semantics of TAEMS plan and schedule can be detailed. A TAEMS plan is a set of methods that can be chosen so that when they are executed within stipulated time, the task group in consideration will accrue some quality. This set of method is derived after following all task-subtask relation constraint, and hard interrelationship constraints. Since many such plans can exist it essentially means the agents have several alternative ways of doing things to accomplish the goal task. Due to the fact that a task structure is a DAG instead of a tree, this problem is a NP-hard problem with complexity of $O(n^n)$, where n is the number of nodes in the DAG.

A TAEMS schedule consists of a sequence of activities (or a plan) with the addition of an intended start time associated with each of these activities. These take care of all the temporal constraints on the nodes.

Each agent will have its own initial schedule which then needs to be coordinated with other agents to synchronize the schedules. The aim of this thesis is to compute the initial schedule for the objective task structure. So, this thesis explains how to compute various possible set of methods and assign them start times so that the task group could be accomplished with some quality.

Constraint Programming(CP)

This thesis utilizes constraint programming concepts to solve the TAEMS scheduling problem. This section provides some simplified concepts of constraint programming.

As mentioned in [1] Constraint programming is a problem solving paradigm which establishes the distinction between precise declarative description of constraints that define the problem and the algorithms and heuristics that enable selection and cancellation of decisions to solve the problem. It is based on strong theoretical foundation and deals with constraints and constraint solving. To illustrate the nuances of CP and show simply how constraint processing in a Constraint Programming Language (CLP) works it can be differentiated from traditional programming languages even OO programming languages. If in a traditional programming language the statement, X = Y + Z holds, it is just an assignment statement and computes X based on the value of Y and Z. If Y and Z are not instantiated, this will result in an error. It is the burden of the programmer to make sure the relationship is maintained and to find objects which satisfy them. Whereas in a CLP this can be specified even if none of the variables are instantiated. The burden to maintain these explicit relationships and ‘constraints’ is solely on the

underlying implementation of the CLP. The programmer need not bother about the low level implementation details of the CLP. When X = Y + Z holds, this is just input as a constraints, where the possible set of values for X, Y and Z are limited. When this constraint is enforced, the values are pruned from the known domain of X, Y and Z to arrive at new domains. If any of the domains become empty, it means the constraint is not satisfiable, meaning that for no value of X, Y and Z, X can be equal to Y plus Z. Another good thing about this approach is that, when one wants to compute the value of Y instead of X, in traditional programming language one need to specify an assignment statement separately, Y = X – Z. But in CLP the same statement X = Y + Z can tell us the possible values for any of the variables X, Y or Z. It just relates these three variables forming a constraints, so X = Y + Z is same as Y = X – Z or Z = X – Y. To show how the pruning occurs lets consider the constraint X = Y. The knowledge of the domains of the variables is required beforehand. Say X can take values from the set {1 , 2 , 3 , 4 , 5} and Y can take values from the set {1 , 3 , 5 , 7}. When X = Y is enforced, the domain of X reduces to {1, 3, 5} and Y to {1, 3, 5}. The values that do not satisfy X = Y are pruned from the domain. Graphically this is shown in Figure 8. Applying constraint X = Y, the values that X and Y can take reduce to the shaded region as shown in Figure 9.

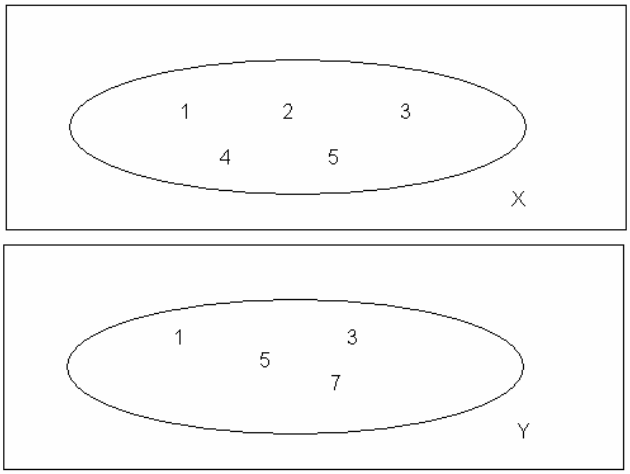


Figure 8. Initial Domain for X and Y

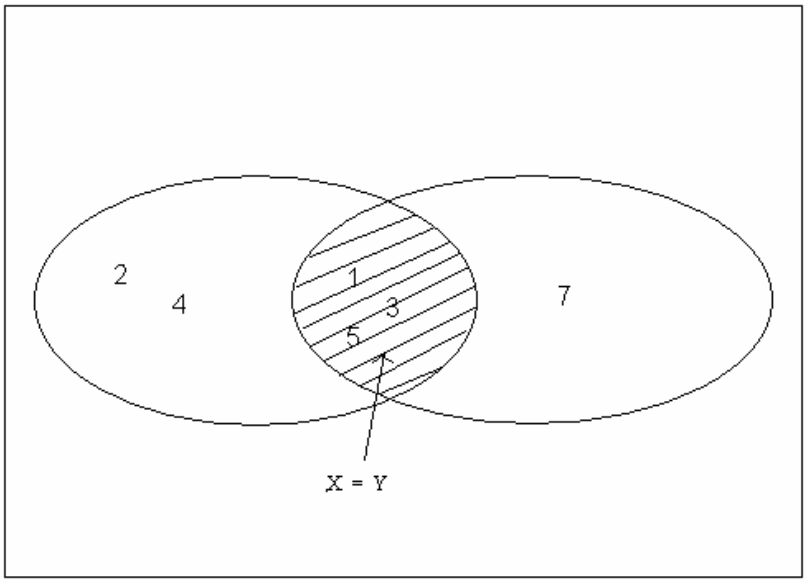


Figure 9. Domain after applying X = Y.

Constraint Satisfaction Problem (CSP)

A Constraint satisfaction problem [2] is a constraint problem in which the goal is to find a consistent assignment of values to variables that satisfies all the constraints. CSP is formally defined by a finite set of variables, the set of values that these variables can take and the

constraints which restrict the combination of values that these variables can take. Solving a CSP is equivalent to finding an assignment for all variables such that all the constraints are satisfied. To find a consistent assignment for the variables, the constraints are repeatedly applied to the domain of the variables to remove the values which are not consistent with the constraint store. This process of repeated application of constraints is called propagation, and the domain is said to be 'pruned'. This process basically removes the inconsistent values from the domain of the variables. To illustrate CSP modeling, consider a crypto-arithmetic problem. Constraint Logic Programming (CLP) can be used to model the problem as a CSP and solve it [2].

Consider the puzzle: S E N D

+ M O R E

= M O N E Y

To find what different digits these alphabets represent, one will require to put a value of each of these alphabets and to see if it works. That will involve a huge amount of processing and implementation of a search algorithm in traditional language.

Whereas in a CLP, this problem can be simply modeled as (ECLiPSe CLP code):

```
puzzle(List) :- List = [ S, E, N, D, M, O, R, Y ], List :: 0..9,
constrain(List), labeling(List).
constrain(List) :- S ## 0, M ## 0, alldifferent(List),
                1000 * S + 100 * E + 10 * N + D
+               1000 * M + 100 * O + 10 * R + E
#=   10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```

The code above means there is a user defined predicate called puzzle(List) which is the goal of the problem. The variables of the problem are added to a list structure. The initial domain of the variables are from 0 to 9 specified as List :: [0..9]. Actual propagation of variable values is

achieved by the built-in predicate ‘labeling(List)’. This assigns the variables a particular value from their domains and tries to see if the assignment is consistent. Constrain is another user defined predicate which models the actual problem. Alldifferent(List) is a built-in predicate which enforces the fact that all the variables assume unique value so that none of them have same value.

This code is sufficient for getting the solution for the puzzle given the goal puzzle(List) i.e. S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8 and Y = 2. The onus of going through the search algorithm to search for a solution is on the CLP Engine and not on the programmer.

Hence, a CLP is a programming language which lets the programmer define the relationship between objects and leaves the maintenance of these relationships to the underlying implementation of the language. It has been successfully used to model complex problems by specifying the constraints between the variables [20]. As a result, any changes in the model can be incorporated by adding and/or removing constraints. This approach is not devoid of drawbacks. Controlling search is still an active research area and there are few generic, high performance techniques available to facilitate search. Another drawback is the lack of efficient debuggers and built-in debugging facility.

Constraint Programming is being successfully used to solve difficult multi-faceted combinatorial problems like those in timetabling, job scheduling and routing. These problems are particularly difficult to tackle using conventional languages because they involve searching through a possibly exponential sized solution space in order to get a solution.

Constraint Solving Techniques

Numerous methods are available for solving constraints. Common among these are simplification, optimization, bound propagation techniques and integer programming techniques

[2]. Simplification is used to make the implicit information apparent by replacing a constraint by another constraint which has a simpler form. Optimization techniques are used when there is a need to find the “best” possible solution. This might require minimization or maximization of an expression. There are also consistency based techniques like arc and node consistency developed by the AI community. These have been widely used to solve scheduling and routing problems. Two techniques that are often used by a solver are: Constraint propagation and Constraint distribution.

Constraint propagation is achieved by constantly adding information about the variables to a constraint store [5]. The store contains information about the values of the variables as a conjunction of the basic constraints [3]. This helps to narrow down the domain of the variable. So if a variable A has domain [1..10], and a constraint A > 7 is added to the store, it propagates to reduce the domain of A to [8..10].

To explain propagation, let us consider the following example. Let us take two constraints for [X, Y] = [0..9]

equation(1): X + Y = 9 and

equation(2) 2X + 4Y = 24

The store initially contains the above domain information. Equation 1 does not do any pruning initially but equation 2 prunes the domain to

X = [0..8] and Y = [2..6].

This happens because for X = 9, no value of Y can satisfy equation 2. Similarly for Y = [0, 1, 7, 8, 9], no value of X satisfy equation 2. Now equation 1 constraint is applied again which reduces domain of

X = [3..7]

but does not change domain of Y. Equation 2 makes it

X= [4..6] and Y = [3..4].

Applying Equation 1 again,

X=[5..6] and Y = [3..4].

Final Equation 2 application, fixes the domain to unique values.

X = 6 and Y = 3, which is the solution to the above simultaneous equations.

The constraint store status is graphically shown in Figure 10.

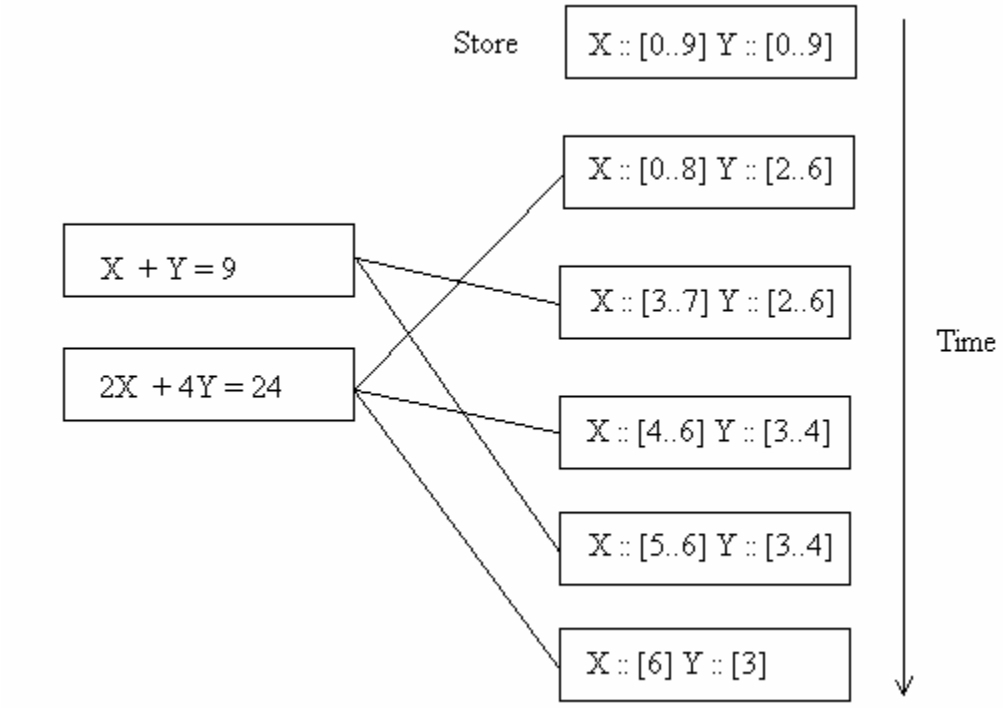


Figure 10. Constraint Store Status

Propagation can be either domain propagation or interval propagation. In interval propagation, only the domain bounds are changed not the all values. Hence, for A * B =8, [A, B] = [0..10], interval propagation will reduce domain to [A,B] = [0..8], Whereas domain propagation will

reduce the values to [A, B] = [1, 2, 4, 8]. Domain gives better results but is computationally more expensive.

Search Techniques

Constraint propagation alone is not enough to solve problems and it is not complete. This happens in cases where propagation can neither reduce the domain to a valuation domain nor proof it to be inconsistent. After this point, search techniques are used. Search proceeds by looking for a solution in a search tree, which is a representation of the search process. The search space is a set of all possible assignments and hence grows exponentially with the number of variables. The root of the search tree represents the state of the search process at start. Leaf of the tree denotes termination with success (a solution) or with failure (no consistent solution). Intermediate nodes in the tree represent the intermediate states. Designer can control the shape as well as the exploration of the search tree [6]. The search process can be divided into refinement based method or repair based methods [2] [3]. Refinement based methods are common, where each variable is assigned a value until a complete solution is found or a constraint is violated. The method works by selecting an unassigned variable from the set of variables. A value is assigned to the variable from its domain. If a constraint is violated, search backtracks to the last distribution point and tries an alternate path. This process repeats for all the unassigned variables. Repair based methods start with a complete assignment and on encountering a violation, the assignment is repaired. This repair is done by assigning different values to one or more variables (which had an inconsistent assignment initially) till a solution is reached. One heuristic that can be applied is to select the variables that are known to violate the maximum number of constraints. This technique is not complete.

The constraint programming language used for this thesis is ECLiPSe CLP [6] version 5.3. It is a Prolog based system and readily available for teaching and research purpose. Due to its Prolog like syntax it is easy to follow and learn. It uses the solving techniques mentioned above. It has provisions for both repair based and refinement based search. It has many programming constructs which makes it usable as a programming language. More information on the ECLiPSe CLP can be found on the user manual [6]. The most common solver library used is the finite domain (FD) library where the domain of the variables has integer values the default domain being [-10000000, 10000000]. The solver normally used here is the FD library.

The search tree is explored in ECLiPSe by default in a depth first manner shown in Figure 11. Search process backtracks only on encountering a failed or success state (leaf nodes).

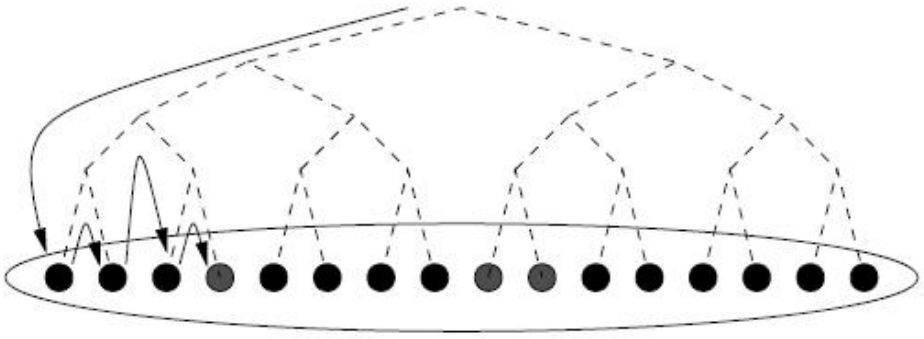


Figure 11. DFS Search Tree

Variable and value selection can also reshape a search tree as shown in Figure 12.

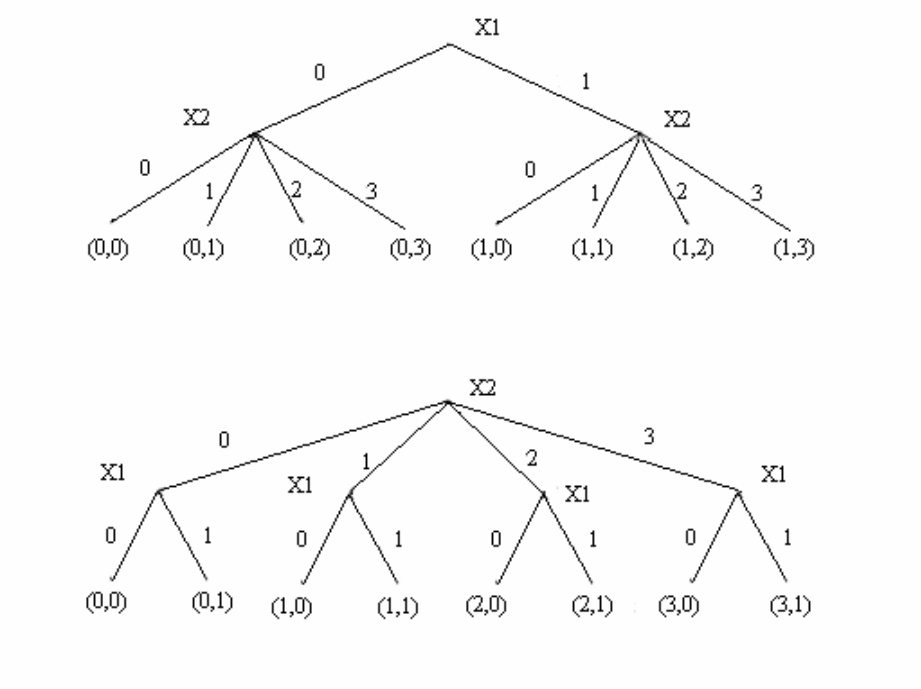


Figure 12. Variable Selection

As shown in Figure 12 the shape of the tree changes depending on whether variable X1 is fixed first or variable X2 is fixed first. ECLiPSe provides support for both complete search techniques and incomplete search techniques. The complete search is done using a construct *labeling,* which basically traverses the whole search tree and finds all solutions.

The default labeling search can be modified to use different heuristics for the search depending on the problem. Other search techniques provided by ECLiPSe are:

1. Bounded backtrack search: - limits the search by the number of backtracks specified in the input.
2. Credit search: - Where the number of non-deterministic choices is limited beforehand.
3. Timeout search: - The time limit is specified after which the search stops.
4. LDS search: - limited discrepancy search [4].
5. Combined LDS and bounded backtrack search.

BBS, credit and combined BBS and LDS have been used for the planning problem here.

Another feature of ECLiPSe is its interfacing and embedding support which has been exploited in this thesis. It can be easily embedded as a solver in many other programming languages like C, C++, Tcl and Java. ECLiPSe terms [6] can be easily manipulated in Java and vice versa. ECLiPSe goals can be executed directly from Java and solutions stored in Java. Different kinds of connections can also be established between the two. There is generally a direct correspondence between Java and ECLiPSe Data Types like integer, list, float, string, atom, and compound. The ECLiPSe engine here is a dynamically loaded native library in the Java Virtual Machine. This means in Embedded Java-ECLiPSe connection, ECLiPSe is not a separate process and shares memory heap/stack and other resources with Java. This memory size can be set while instantiating the ECLiPSe Engine from Java. UML deployment of the embedded ECLiPSe is shown in Figure 13.

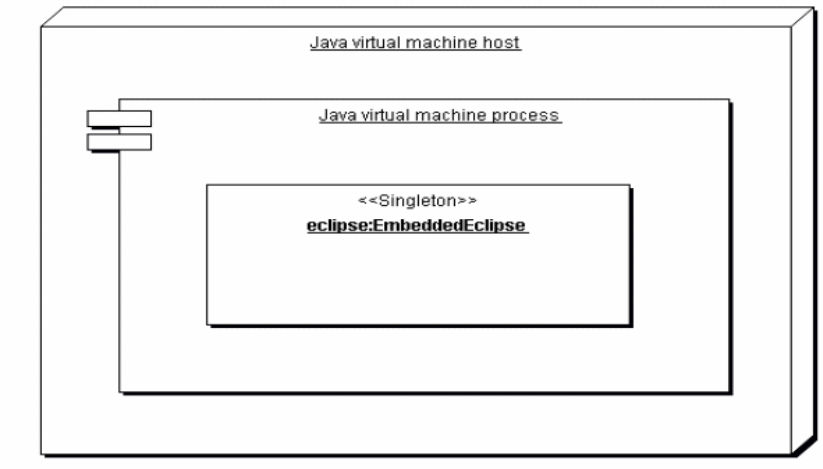


Figure 13. Embedded ECLiPSe in JVM [6]

ECLiPSe also has built-in modules for scheduling problems like *cumulative*, *edge_finder*, *edge_finder3*. But these libraries cannot be used for TAEMS scheduling as it is more complex

due to its hierarchical and interrelationship structure. Hence each constraint has to be formulated separately instead of using the built-in construct.

Alternative Constraint Solvers:

1. Mozart [5]: is a development platform for intelligent distributed applications and is based on the Oz language. Oz is a high level programming language that is designed for concurrent, intelligent, networked, soft real-time, parallel, interactive applications. It allows the developers to program and control search strategies.
2. CHIP [18]: is a complete environment for the design and development of decision support systems.
3. Choco [22]: is a free Java based library for solving CSP problems. It utilizes event-based mechanism with backtrackable structures. It utilizes imperative constraint programming and is available as a library.
4. ILog solver [19]: is a suite of 3 libraries implemented in C++ - ILog solver, scheduler and planner. It is also imperative constraint programming language and provides a powerful solver.

CHAPTER III

AN APPROACH TO TAEMS SCHEDULING USING CONSTRAINT PROGRAMMING

This chapter provides the details of the modeling and encoding techniques used to solve the TAEMS planning and scheduling problem and to generate a sequence of methods and their start times whose execution might result in a non-zero quality at the task group level. The view in consideration here is the objective task structure of the problem. To draw relevance with the discussion in the background section, this will essentially entail converting the TAEMS planning and scheduling problem into a finite domain CSP, solve it using ECLiPSe and convert the solution back to a schedule. These processes have been discussed in detail in this chapter. To compute the final schedule this problem has been solved in two parts – first plans containing sequence of methods have been obtained and second execution start time have been assigned to these methods in the plans (i.e. the plans are scheduled). TAEMS has a Java implementation [21] which makes it easy to access various components of a TAEMS structure. This representation is the input to our process.

Planning Encoding

A TAEMS problem is a tree/graph like task structure containing root task, subtasks, executable methods and interrelationships, each having some attributes. To find a sequence of methods which if executed in sequence will achieve the root task requires searching through the task structure and hence an efficient implementation of a search algorithm specific to the TAEMS problem. For pure tree like structures this is a simple problem and traversal and search can be done in polynomial time. But due to presence of complex interrelationships in a TAEMS

structure, the task structure actually assumes the shape of a graph and hence traversal and search takes O (n^n) time and huge memory requirements making the problem an NP-hard problem. This is very much possible in a traditional programming language like Java or C++ using sophisticated heuristics and search methods but if the problem specification changes or increases in complexity, it will be difficult to maintain the algorithm and maintenance may prove time-consuming. Instead of implementing a complex algorithm, the problem can be tackled using CSP approach. To convert the TAEMS planning problem into a CSP, first a domain needs to be chosen as per the attributes involved. This is required to determine which solver library will be used to solve the problem. Next a simple problem model will be constructed from TAEMS. Constraints are formulated and search heuristics are formulated to improve performance. ECLiPSe CLP [6] allows us to define variables, domains and constraints. The process of encoding requires the transformation of the generic problem representation to the input format of an ECLiPSE specific solver. The converse of this is the decoding process which maps the result of a solver to the actual problem solution.

Domain Selection

CP is a widely used term for various solver technologies common in declarative problem specification. They are characterized by the range of possible values (or domains) of variables, and the types of constraints that can be handled. In ECLiPSe CLP, the various solvers available are IC, RIA, FD, eplex, fdplex etc [6]. Each of these is specialized for particular problem domains. For example, RIA is for problems where variable domains are represented as real ranges or interval of real values. FD is for finite domain problems where a variable can assume finite integer values only. IC is an integrated solver capable of handling both real values and integer values. One of these solver technologies must be selected first. The most frequently used

solver is the finite domain solver and is present in almost all CPs. They are capable of handling Booleans and integers, but the range of values an integer can take is limited. This is a limitation of the ECLiPSe FD solver but usually not an issue. The domain of choice in our case is the FD solver. This choice will become clearer once modeling and constraint formation is discussed.

Modeling

A CSP consists of variables, domains and constraints over those variables. A model describes how the unknown factors in the problem are specified by the domain variables. In lucky cases there can be an obvious natural mapping, especially for FD domain but in other cases mapping can be a complex task. In the context of TAEMS problem, the mapping is not entirely straight forward but not very complex either.

In TAEMS planning, the point of interest is to know which methods can form a plan. The main entities present in the objective TAEMS structure are:

1. Tasks
2. Methods
3. Interrelationships
4. Agents

Each Task is modeled as a variable in the FD domain. Since label of a task uniquely identifies the task, the variables have been named after the task labels. This ensures uniformity as well as integrity.

Although Tasks are not part of the TAEMS plan, they need to be modeled because of the fact that they introduce additional constraints and simplify the process of constraint formation but do not increase the complexity of the search process. The values of these variables are not of particular interest but are used to specify constraints.

Each Method in the structure is also modeled as a variable in the FD domain. Since label of a Method uniquely identifies the method, the variables have been named after the method labels. The values of these variables will ultimately determine whether the method will be present in the plan. These are the variables whose values are of interest.

Agents are not considered in a plan because they will not influence the planning process or selection of the methods. They are just supposed to help in the execution of these methods. The interrelationships also do not require a variable representation since the value for an interrelationship are not of interest. The semantics of the NLEs are however taken care of by constraints formulation which is discussed next. Hence, for a simple structure like Figure 14 in the following page, the variables are shown in bold.

Another issue is the domain of values these variables modeled above can assume. Since our dealing here is with FD constraints, the values should be finite integers. The final result that is required here is whether a method is included in the plan or excluded from the plan. This is a logical decision and the method variables assume Boolean values. Hence, if value of a method variable is 1, it is in the plan and will be executed if scheduled and helps in gathering quality for the task group.

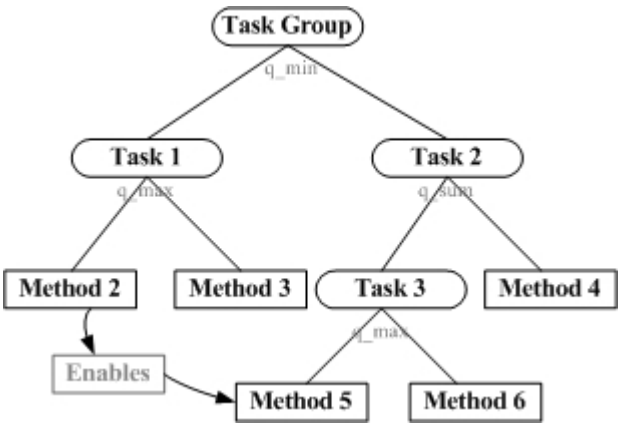


Figure 14. CSP Variables shown in Bold

If it is 0, it is not included in the plan and does not contribute any quality for that plan. Hence the Method variables assume Boolean values. So domain of the Method variables in the planning context is [0..1]. Similarly, for the task variables, the domain is [0..1], which means whether they have to be accomplished or not in order to accomplish the root task in a successful plan. For example, for Figure 14, the possible plans are:

- Method 2, Method 5
- Method 2, Method 6, Method 4 etc…

Which means the values of these methods will be 1 and the remaining values will be 0.

Generating Constraints

Generating constraints requires enforcing relations between the domain variables depending on the problem specification. This enforces pruning of the domains before search. Modeling influences this mapping between the problem and the domain constraints. The simpler the model is the more direct the correspondence between the problem and constraints. CP expertise makes this process better by choosing correct amount of redundancy and optimizes performance.

To correctly enforce all the possible relations, the requirement is to relate the variables modeled above. Finally the solver is invoked for values of the method variables.

1. The root task always needs to be achieved. This is the basic requirement of the whole planning problem i.e. to select methods in order to accomplish root task. There may be many root tasks in the objective view but only one task group. The root task here refers to the task group. To enforce the fact that a root task is always achieved, its value is forced to be 1. Hence for Figure 14 task group = 1. This instantly prunes its domains from [0, 1] to [1].

2. To take care of the Task- Subtask relationship in the structure, the parents and its children need to be related based on their quality accumulation functions. The following QAFs have been handled :

   a) Max – If the QAF of a task is Max, it means that it will take the maximum value of any one of its subtasks. To model it as a constraint so that the subtask chosen somehow makes sure that the parent is always achieved, the essential thing is to enforce a constraint between the parent task variable and its children as an OR constraint. This ensures that at least 1 or more subtasks under a MAX QAF task are performed. So, in Figure 6,

      Task_group <=> (Method1 \/ Method2 \/ Method3).

      The possible combination of values for the child nodes are (1,1,1), (1,1,0), (1,0,1),(0,1,1),(1,0,0),(0,1,0),(0,0,1),(0,0,0).

      Except (0,0,0) all the above will lead to Parent_task = 1.

   b) Min – If the QAF of a task is Min, it means that the parent will take the minimum value of all its subtasks. To model this as a constraint so that the subtask chosen always makes sure that the parent is achieved, it is necessary to enforce an AND constraint between the children. This ensures all the children must be achieved in order to achieve the parent. Figure 7,

      Task_group <=> (Method 1 /\ Method 2 /\ Method 3).

      The possible combination of values for the child nodes are (1,1,1), (1,1,0), (1,0,1),(0,1,1),(1,0,0),(0,1,0),(0,0,1),(0,0,0).

      Only for (1,1,1) the Parent_task = 1, for all other values it is 0.

c) Sum – If the QAF of a task is Sum, it means that the parent will take the sum of the quality of its children. To make sure such a parent has quality at least one of its children should be achieved. To model it as a constraint so that the subtask chosen always makes sure that the parent is achieved, the essential thing is to enforce a constraint between the parent task variable and its children as an OR constraint. This ensures that at least 1 or more subtasks under a SUM QAF task are performed. In Figure 5,

Task_group <=> (Method 1 \/ Method 2 \/ Method 3).

This encoding is same as QAF Max because the objective here is just to find all possible ways to achieve non-zero quality at root task. Both QAF Sum and Max do not require all sub-nodes to execute before a parent task can accumulate quality. Hence it is possible to accumulate quality when at least one or more of the sub-nodes have positive quality in both cases. The possible combination of values for the child nodes are (1,1,1), (1,1,0), (1,0,1),(0,1,1),(1,0,0),(0,1,0),(0,0,1),(0,0,0).

Except (0,0,0) all the above will lead to Parent_task = 1.

d) Syncsum – If the QAF of a task is Syncsum, only those tasks that start at the same time as the parent accumulate quality. To make sure such a parent has quality at least one of its children should be achieved. To model it as a constraint so that the subtask chosen always makes sure that the parent is achieved, the essential thing is to enforce a constraint between the parent task variable and its children as an OR constraint. This ensures that at least 1 or more subtasks under a Syncsum QAF task are performed. So, in Figure 5,

Task_group <=> (Method 1 \/ Method 2 \/ Method 3).

This encoding is also equivalent to QAF Max and Sum.

The possible combination of values for the child tasks are (1,1,1), (1,1,0), (1,0,1),(0,1,1),(1,0,0),(0,1,0),(0,0,1),(0,0,0).

Except (0,0,0) all the above will lead to Parent_task = 1.

Other QAFs as mentioned in the TAEMS white paper can be similarly taken care of but not handled in this thesis. Like QAF 'exactly one' can be enforced by XOR constraints. QAF 'All' can be handled similar to QAF 'Min' etc.

3. If any of the parent above has only one child, then irrespective of the QAF, the following constraint is enforced,

   Parent #<=> child,

   This bi-implication means the child task must be accomplished to accomplish the parent task. Since QAF is only a means to combine the children quality, it will not have any significance (for planning) in case of a parent task with a single child.

4. The interrelationship of Facilitates and Hinders do not need to be considered as they don't influence the planning process. This is because they just enforce soft constraints and not hard ones since this thesis aims to find working plans and not 'good' or 'bad' plans. The Enables NLE is modeled in the following way:

   a) Enables interrelationship enforces a hard constraint by specifying that in order to start the target node; the root node must have accumulated some non-zero quality. If the target node is started and the source node has zero quality then the target node fails i.e. accumulates zero quality. To capture this as a planning constraint an implication relation is needed between the source and the target node where the target node is achieved only when the source node is achieved.

      Hence in Figure 14, Method 5 #=> Method 2

General structure is: targetnode #=> sourcenode

5. The last constraint requirement enforces that a method is included in the plan only if all its parents are achieved provided it is not a source node of an interrelationship. This is enforced for all methods. General structure is,

   Method => (parent /\ Subparent1 /\ Subparent2. In Figure 14,

   Method 5 #=> (Task Group #/\ Task 2 #/\ Task 3).

Interfacing and Solving

The software developed for this thesis to solve the planning problem has a Java front-end and ECLiPSe solver back-end. Most of the data pre-processing is done in Java to formulate the constraints whereas ECLiPSe is used to find the solutions making use of both traditional programming and CP. The Eclipse solver does not do any data pre-processing hence saving time during the actual solving process. Once the solver finds a solution to the planning CSP, control is passed back to Java to interpret the solution and convert it to a TAEMS plan. Since ECLiPSe goals can be executed directly from Java, no sophisticated interface technology is required to transfer data between the two.

ECLiPSe is available in the form of a linkable library, and a number of facilities are available to pass data between the solver and Java to make the integration as close as possible. Java programs can be written to [6]:

- Manipulate ECLiPSe terms and other data
- Execute goals in ECLiPSe and use the result of this computation in Java.
- Transfer data between Java and ECLiPSe using queues.
- Initialize and terminate different kinds of interfacing connections.

The Javadoc for API reference is available in [6]. The constraints formulated above are saved to an ECLiPSe file, and the ECLiPSe goal is executed from Java. The general correspondence between ECLiPSe and Java datatypes is shown in Figure 15.

ECLiPSe Data Type	Java Class/Interface
atom	Atom
Compound	Compound Term
Integer	java.lang.Integer, java.lang.Long
List	java.util.Collection
float	java.lang.Double, java.lang.Float
string	java.lang.String
variable	null

Figure 15. Java ECLiPSe Datatype Mapping [6]

There are 3 approaches to interface Java with ECLiPSe:

1. Embedded ECLiPSe
2. Out-of process ECLiPSe
3. Remote ECLiPSe.

The connection suitable to be used in our problem is Embedded ECLiPSe since goals can be executed directly and result can be processed using RPC (Remote Predicate Call) method. This also has efficient data transfer mechanism since Java and ECLiPSe share the same memory stack/heap. Non-determinism is taken care of in ECLiPSe, so if there are more than one plans, ECLiPSe will collect all these plan and pass it to Java as a single List.

Search Heuristics

Depending on the constraint system used, search may or may not be needed. For problems with limited complexity, an exhaustive search does not take much resource or time. But as problem complexity increases, it might be essential to employ search heuristics. In-built search options like Branch and bound search, Credit search, time-out search are present in almost all solvers. They can be tweaked and customized for use in any search problem.

In our case, the default search heuristics is branch and bound search (BBS) [Code in Appendix B] which counts the number of backtracks and curtails the search when this number reaches a pre-set value. This has proved to be a good heuristics to limit search in case of huge problems with thousands of plans. For the purpose of this thesis, initial backtrack limit is set to 100. This number can be changed to any integer value acceptable to ECLiPSe.

Other incomplete search heuristics employed are credit-search and LDS + BBS. Credit search is a tree search method where the number of nondeterministic choices is limited a priori. This is achieved by starting the search at the tree root with a certain integral amount of credit. This credit is split between the child nodes, their credit between their child nodes, and so on. A single unit of credit cannot be split any further: sub-trees provided with only a single credit unit are not allowed any non-deterministic choices, only one path though these sub-trees can be explored, i.e. only one leaf in the sub-tree can be visited. Sub-trees for which no credit is left are pruned, i.e. not visited. Initial credit assigned in case of credit search for this thesis is cube times the number of methods. This is a good approximation of the search heuristic credit limit.

Limited discrepancy search (LDS) is a search method that assumes that the user has a good heuristic for directing the search. The “discrepancy” is a measure of the degree to which the search fails to follow the heuristic. LDS starts searching with a discrepancy of 0 (which

means it follows the heuristic exactly). Each time LDS fails to find a solution with a given discrepancy, the discrepancy is increased and search restarts. In theory the search is complete, as eventually the discrepancy will become large enough to admit a solution, or cover the whole search space. In practice, however, it is only beneficial to apply LDS with small discrepancies. In this thesis, LDS has been used in combination with BBS and the search starts assuming that the initial plan has all the methods. It proceeds from here to find all the plans limited by the number of backtracks. More on the search heuristics experimentation are provided in results section Chapter IV.

Decoding

Finally the results list obtained from ECLiPSe solver is returned to Java and processed to construct the plan i.e. sequence of methods. There is a direct correspondence between this list and the plans. A value of 1 for the method variables in the list means the method has been included in that plan and a value of 0 means it has been excluded from the plan. All such plans are reconstructed in order and passed to the scheduler one by one to assign a start times for the method executions.

Overall Planner pseudocode:

Input: Taems task structure.

1. Create an ECLiPSe file and include FD domain.
2. Create domain variables for each method and task and declare the domain to be [0..1].
3. Create Problem constraints :
    i) Root task variable value should be 1.

ii) Post QAF constraints :

a) If QAF = max, post ‘or’ constraints among the children

b) If QAF = min, post ‘and’ constraints among the children

c) If QAF = sum, post ‘or’ constraints among the children

d) If QAF = sync sum, post ‘or’ constraints among the children

iii) Post NLE constraints – target task implies source task.

iv) Post Method constraints – method implies all its parents.

4. Declare the search strategy – either full tree search, BBS, Credit, LDS + BBS search.
5. Create an instance of the ECLiPSe engine and execute the plan goal from the RPC method.
6. Decode the solver result to construct the possible plans.

Output: A list of plans, each containing a set of methods.

Scheduling

Once a plan is obtained from the process as described above the temporal constraints mentioned in the model need to be handled to determine when these methods can execute. This section provides the details of the modeling and encoding techniques used to solve the TAEMS scheduling problem and to assign start times to the plan methods so that their execution result in a non-zero quality at the task group. The view in consideration here is the objective task structure of the problem. To draw relevance with the discussion in the background section, this will essentially entail converting the TAEMS scheduling problem into a CSP, solve it using ECLiPSe and convert the solution back as a course of action (or CoA) which has all the methods including

their scheduled execution start times. If a plan cannot be scheduled it is excluded from the list of schedules. It means for no possible time frame, these methods can execute to accomplish the task group (accumulate quality at root). All the schedules have the form:

(method, start time)

Few types of schedules for the TAEMS problem are described below [24] –

- Basic Schedule - is a sequence of methods and start times. It is rigid, in the sense that methods are started at the appointed time regardless of any other state, such as whether or not the previously-started method in the same agent has completed execution.
- Non-conflicting Basic Schedule - is one in which the start times are spaced such that if any method takes its maximum defined execution duration, it will complete before the next method is started. So, it assumes the worst case scenario and schedules pessimistically.
- Conflicting Basic Schedule - is one in which one or more methods might not complete execution before the following method is scheduled to be started. This happens when the methods execute for more than their expected duration.
- Flexible Schedule - is one in which a method may be started within some range of times. Flexible schedules may be conflicting or non-conflicting. They have the form: (method, earliestStartTime, latestStartTime)

This thesis can generate a basic initial schedule. Flexible schedules can also be easily generated from the same framework by making a few changes. The basic assumption while scheduling is that the methods will execute within their expected duration time. So no conflict condition is assumed. Scheduling only considering the methods which have been pre-selected using the planning process described above.

Encoding

To find a start time for the method sequences which when executed will achieve the root task requires enforcing the temporal constraints imposed by the structure. Due to presence of complex interrelationships in a TAEMS structure, off-the-shelf built in schedulers cannot be used. To convert the TAEMS scheduling problem into a CSP, first a domain needs to be chosen as per the attributes involved. This will be required to determine which solver library will be used to solve the problem. Next a simple problem model is constructed in it from TAEMS, constraints are formulated and search heuristics are formulated to improve performance. The process of encoding requires transformation of the generic problem representation to the input format of an ECLiPSe specific solver. The converse of this is decoding which maps the result of a solver to the actual problem solution.

Domain Selection

The domain of choice in our case is the FD solver. This choice will become clearer once modeling and constraint formation is discussed. Since the time range here is considered to be integers, finite domain is a valid choice.

Modeling

The modeling of a TAEMS scheduling problem is almost same as the planning problem except for the domain of the variables since temporal constraints are involved here. These variables now represent the start times of the methods and the tasks now instead of Booleans. The main entities present in the objective TAEMS structure are:

1. Tasks
2. Methods

3. Interrelationships

4. Agents

Each Task which has a method (included in the plan) in the hierarchy below it is modeled as a variable in the FD domain. Since label of a task uniquely identifies the task, the variables have been named after the task labels. This insures uniformity as well as integrity.

Although Tasks are not part of the TAEMS schedule, they are modeled because of the fact that they introduce additional constraints and simplify constraint formation without increase the complexity of the search process. The values of these variables are not used for generating the schedule.

Each Method in the input plan is also modeled as a variable in the FD domain. Since label of a Method uniquely identifies the method, the variables have been named after the method labels. The values of these variables will denote the start times of the methods. These are the variables whose values are used to construct the final schedule.

Agents are not considered to be modeled as variables in the CSP model primarily because they are considered as resources in this process. This thesis handles the presence of agents by enforcing a serialization constraint but does not model agents as variables. The interrelationships also do not require a variable representation since such value is of no interest for the purpose of scheduling. The semantics of the NLEs are however handled by constraints formulation which is explained in the following paragraph. In Figure 14, the variables are shown in bold. They denote the node start times.

Another issue is the domain of values these variables modeled above can assume. Since these are modeled as FD constraints, the values should be finite integers. So the requirement is to

have a range of all possible start times for these nodes. For this purpose, evaluation of the time attributes provided for the nodes is required.

1. Methods have -

a) Earliest start times – if a method is executed before this, it will not accrue quality.

b) Deadlines – if a method is executed after this, it will not accrue any quality.

c) Expected duration – the expected run time of the method.

Hence the latest time where a method can start is

Latest start time = Deadline – Expected Duration

So for the method start time, variables range of domain values is

[Earliest start time .. Latest start time]

While calculating the earliest start times and deadlines following guidelines are used.

For a node n [24],

$$Earliest-start^{*}(n) = \max(earliest-start(n), \max(earliest-start^{*}(T))),\ T \in supertasks(n)$$

Similarly,

$$deadline^{*}(n) = \min(deadline(n), \min(deadline^{*}(T))),\ T \in supertasks(n)$$

So, the actual calculation is done using propagated times.

2. Tasks have –

a) Release Time – before which they cannot start.

b) Deadlines – after which they stop accruing quality.

So for the method start time, range of variable domain values is

[Release time .. Deadline]

All the times values are propagated. Hence, the possible values of task and method start time variables are obtained though only the method start time variable values affect the scheduling.

Generating Constraints

To correctly enforce all the possible relations, the requirement is to relate the variables modeled above. Finally, solution is obtained for the values of the method start time variables.

1. The parent task starts as soon as any of its sub-tasks start. Hence for a Task having only one child the start time of the parent-child will be same. For parent node having more than one child, the parent start time will be earliest of its children start times. For Figure 14, the ECLiPSe constraints are formulated as:

   minlist([Task1, Task2], Task Group),

   minlist is a built-in predicate to take the minimum value of the list.

2. To take care of the Enablement Disablement interrelationship in the structure, start time constraints need to be enforced among the related variables. The idea here is that all enabling nodes should complete before the target enabled node can start execution. All the enable relations on the input plan methods (propagated from any enables from any of the parent tasks as well) are considered and the source of these NLEs is traced. The source Task should complete before the planned methods can start. The source task completion means all its scheduled methods or subtasks have been completed. In Figure 14, The enablement constraints are:

   Method 2 + Method 2_duration #<= Method 5

   In general, sourcemethod + sourcemethod_duration #<= targetmethod

   Similarly disablement constraints are:

   targetmethod + targetmethod_duration #<= sourcemethod

3. Serialization constraints for multiple agents – In a multi-agent environment, there are several agents that own various tasks and methods and are responsible for their execution. These

agents are unaware of the big picture of an objective view. Serialization constraint needs to be enforced in such cases to make sure an agent can execute at most one method at a time. In Figure 14, if both method2 and method3 belong to agent1, constraints can be formulated as:

(Method2 + Method2_duration #<= Method3) #\/

(Method3 + Method3_duration #<= Method2)

The formulation above ensures that method 3 cannot start unless method 2 completes or method 2 cannot start unless method 3 completes. Hence, the start times of the methods are fixed such that no agent will have more than one method to execute at a time. Similar serialization constraints are formulated for all plan methods sharing agents.

4. The last thing of consideration is the sync sum QAF which enforces a hard constraint on the start times of the parent task with sync sum QAF and its children. In a task structure with a parent task having QAF syncsum and children child1 and child 2, constraints are parent #= child1 and parent #= child2, hence forcing the start times of parent, child1, and child2 to be same. There is an exception to this case. If a child is a source node for an enabling constraint and the enabled node is present in the plan, this parent child constraint is not applied.

Interfacing, Solving and Decoding

The interfacing setup for scheduling is same as described in the planning process. There might be more than one schedule for a plan which signifies that methods start times are not rigid. In this thesis, the schedule with the minimum average start time for all methods is taken. The labeling predicate in ECLiPSe handles this requirement. Non-determinism is avoided here by passing back a single schedule with minimum average start time.

The schedule thus obtained is the final schedule for the plan. Hence, if the methods are executed as per the start times specified in the schedule, there should be some positive quality

accumulated by the root task. Java checks each schedule before adding it as a final schedule for positive quality at root. Finally the schedules are stored as a XML data file.

Overall Scheduler pseudocode:

Input: A Taems structure and a plan.

1. Create an ECLiPSe file and include FD domain.
2. Create domain variables for each input plan method and associated task
3. Associate appropriate domain with the variables after propagating all time values,
    i) For methods, domain is

       [Earliest start time .. Deadline – Expected duration].

    ii) For tasks, domain is

       [Release time .. Deadline]
4. Create Problem constraints :
    i) Parent starts as soon as any of its child starts.
    ii) Enabling tasks finish before enabled tasks.
    iii) Agents can process one constraint at a time.
    iv) Tasks with QAF syncsum, start at the same time as its children.
5. Declare the search strategy. Labeling is used here.
6. Create instance of ECLiPSe engine and execute the scheduling goal from the RPC method.
7. Decode the solver result to construct the possible schedules.
8. Verify that the initial schedule returned would indeed produce positive quality at root task.

Output: The final schedule for the given plan with minimum average start time.

CHAPTER IV

CASE STUDY

In this chapter, a few examples of TAEMS structures of varying complexity are illustrated and the CSP method is used to generate schedules for these structures. The output is a set of methods and start times. A comparison of the result of using various search methods is also presented here along with analysis of the results.

Example 1: p11f1 Taems structure [23]. This is very simple example with one root task, 6 tasks, 6 methods, 6 enablement relations and 4 agents shown in Figure 16.

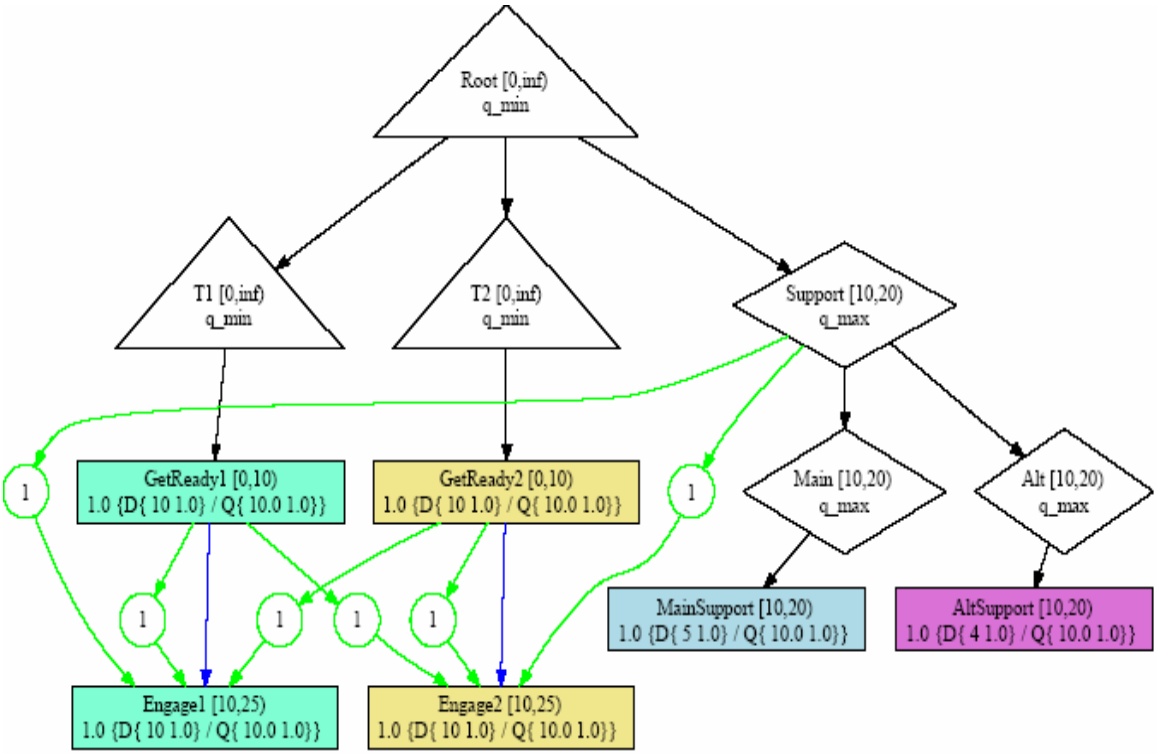


Figure 16. Example p11f1 [23]

Triangles denote tasks with Min QAF, diamond shaped boxes denote tasks with Max QAF, hexagon shaped boxes denote task with QAF Sum. Rectangles denote methods. Methods visible to a particular agent have same color. Solid arrows denote task-subtask relation. Arrows connecting two or more methods denote sibling relation (child of the same parent task). Arrows with circle connectors denote enablement.

The above Figure 16 is a TAEMS Task structure with the relevant details shown in Table 4, Table 5, Table 6 and Table 7.

Table 4. Agents Table Example 1

Agent	**Method color**	**Methods Owned**
Team1	Green	GetReady1, Engage1
Team2	Yellow	GetReady2, Engage2
Team3	Blue	MainSupport
Team4	purple	AltSupport

Table 5. Task Table Example 1

Task	**Subtasks**	**QAF**	**Release time**	**Deadline**
Root	T1, T2, Support	Min	0	infinity
T1	GetReady1, Engage1	Min	0	infinity
T2	GetReady2, Engage2	Min	0	infinity
Support	Main, Alt	Max	10	20
Main	MainSupport	Max	10	20
Alt	AltSupport	Max	10	20

Table 6. Method Table Example 1

Method	**Parents**	**Agent**	**Earliest Start**	**Deadline**	**Expected Duration**	**Expected Quality**
GetReady1	T1	Team1	0	10	10	10
Engage1	T1	Team1	10	25	10	10
GetReady2	T2	Team2	0	20	10	10
Engage2	T2	Team2	10	25	10	10
MainSupport	Main	Team3	10	20	5	10
AltSupport	Alt	Team4	10	20	4	10

Table 7. Enablement Table Example 1

Source Node	Target Node
Support	Engage1
Support	Engage2
GetReady1	Engage1
GetReady1	Engage2
GetReady2	Engage1
GetReady2	Engage2

The planning constraints for the above task structure are written in the planning constraints ECLiPSe file shown below:

```
:- lib(fd).
plan(Coalist) :-
Coalist =
[GetReady1,Engage1,GetReady2,Engage2,MainSupport,AltSupport],
Coalist :: 0..1,
[T1, Main, Alt, T2, Root, Support] :: 0..1,
Root #= 1,
Root #<=> (T1 #/\ T2 #/\ Support), T1 #<=> (GetReady1 #/\ Engage1),
T2 #<=> (GetReady2 #/\ Engage2), Support #<=> (Main #\/ Alt),
Main #<=> MainSupport, Alt #<=> AltSupport,
Engage1 #=> GetReady1, Engage1 #=> GetReady2,
Engage1 #=> Support, Engage2 #=> GetReady1,
Engage2 #=> GetReady2, Engage2 #=> Support,
GetReady1 #=> (T1 #/\ Root), Engage1 #=> (T1 #/\ Root),
GetReady2 #=> (T2 #/\ Root), Engage2 #=> (T2 #/\ Root),
MainSupport #=> (Main #/\ Support #/\ Root),
AltSupport #=> (Alt #/\ Support #/\ Root),
labeling(Coalist).
planall(L) :- findall(Coalist, plan(Coalist), L).
```

This file is input to the ECLiPSe solver. The plans generated from the solver:-

Number of plans generated: 3
Plan1: Methods (5) - GetReady1, GetReady2, AltSupport, Engage1, Engage2
Plan2: Methods (5) - GetReady1, GetReady2, MainSupport, Engage1, Engage2
Plan3: Methods (6) - GetReady1, GetReady2, AltSupport, MainSupport Engage1, Engage2

The scheduling constraint ECLiPSe file generated for Plan1 is shown below:

```
:- lib(fd).
:- lib(fd_global).
sched(Coalist) :-
Coalist = [AltSupport, Engage1, Engage2, GetReady1, GetReady2],
AltSupport :: 10..16, Engage1 :: 10..15, Engage2 :: 10..15, GetReady1
:: 0..0, GetReady2 :: 0..0, Root :: 0..25, T1 :: 0..25, T2 :: 0..25,
Support :: 10..20, Alt :: 10..20, minlist([T1, T2, Support], Root),
minlist([GetReady1, Engage1], T1), minlist([GetReady2, Engage2], T2),
```

```
Support #=  Alt, Alt #= AltSupport,
GetReady1 + 10 #<= Engage1, GetReady2 + 10 #<= Engage1,
AltSupport + 4 #<= Engage1, GetReady1 + 10 #<= Engage2,
GetReady2 + 10 #<= Engage2, AltSupport + 4 #<= Engage2,
(Engage1 + 10 #<= GetReady1)  #\/  (GetReady1 + 10 #<= Engage1) ,
(Engage2 + 10 #<= GetReady2)  #\/  (GetReady2 + 10 #<= Engage2) ,
labeling(Coalist).
```

The scheduling constraint ECLiPSe file generated for Plan3 is shown below:

```
:- lib(fd). :-
lib(fd_global).
sched(Coalist) :-
Coalist = [AltSupport, Engage1, Engage2, GetReady1, GetReady2,
MainSupport], AltSupport :: 10..16, Engage1 :: 10..15, Engage2 ::
10..15, GetReady1 :: 0..0, GetReady2 :: 0..0, MainSupport :: 10..15,
Root :: 0..25, T1 :: 0..25, T2 :: 0..25, Support :: 10..20, Main ::
10..20, Alt :: 10..20,
minlist([T1, T2, Support], Root), minlist([GetReady1, Engage1], T1),
minlist([GetReady2, Engage2], T2), minlist([Main, Alt], Support),
Main #= MainSupport, Alt #= AltSupport,
GetReady1 + 10 #<= Engage1, GetReady2 + 10 #<= Engage1,
MainSupport + 5 #<= Engage1, AltSupport + 4 #<= Engage1,
GetReady1 + 10 #<= Engage2, GetReady2 + 10 #<= Engage2,
MainSupport + 5 #<= Engage2, AltSupport + 4 #<= Engage2,
(Engage1 + 10 #<= GetReady1)  #\/  (GetReady1 + 10 #<= Engage1) ,
(Engage2 + 10 #<= GetReady2)  #\/  (GetReady2 + 10 #<= Engage2) ,
labeling(Coalist).
```

Final schedules generated per plan are:

```
Schedule1: GetReady1: 0, GetReady2: 0, AltSupport: 10, Engage1: 14,
           Engage2: 14
Schedule2: GetReady1: 0, GetReady2: 0, MainSupport: 10, Engage1: 15,
           Engage2: 15
Schedule3: GetReady1: 0, GetReady2: 0, MainSupport: 10, AltSupport: 10, Engage1:
            15, Engage2: 15
```

The above initial schedules contain 3 plans along with start times for the methods. Appendix C shows the xml file generated from the solver. The schedules are pictorially represented as Gantt charts in Figure 17, Figure 18 and Figure 19. In Figure 16, the schedules generated by the solver can be verified intuitively. In Schedule1, methods GetReady1 and GetReady2 start at the beginning (time 0) and run for 10 time units. Method AltSupport starts after 10 time units since its earliest start time is 10 and runs for 4 time units. Methods Engage1 and Engage2 cannot start

before 14 time units although its earliest start time is 10 because it is ‘enabled’ by the task Support (which ends after 14 time units) and methods GetReady1 and GetReady2 (both end after 10 time units). So the earliest start time for Engage1 and Engage2 is after 14 time units. Hence if schedule 1 is used to execute the tasks and they execute within the expected duration, a quality of 10 will be accumulated at the root task Root. Similarly, in schedule 3 (all 6 methods), Engage1 and Engage2 can only start when all the other methods have completed since all other methods enable these directly or indirectly.

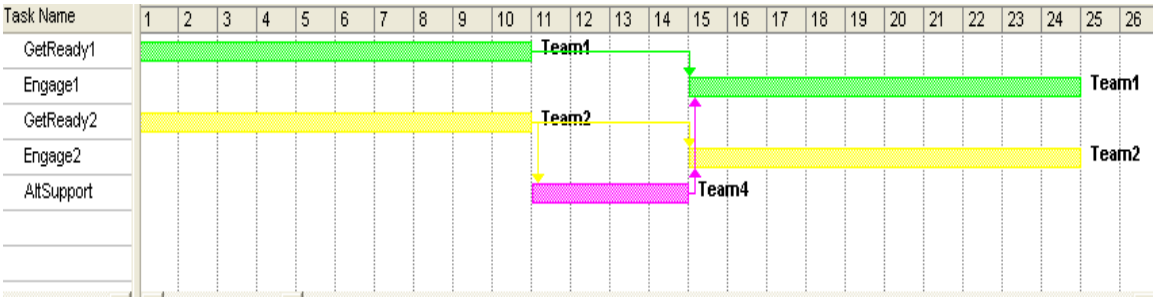


Figure 17. Gantt chart for Schedule1

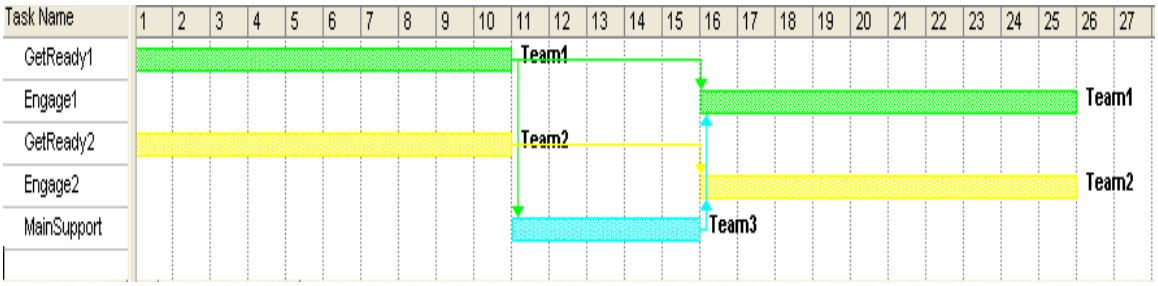


Figure 18. Gantt chart for Schedule2

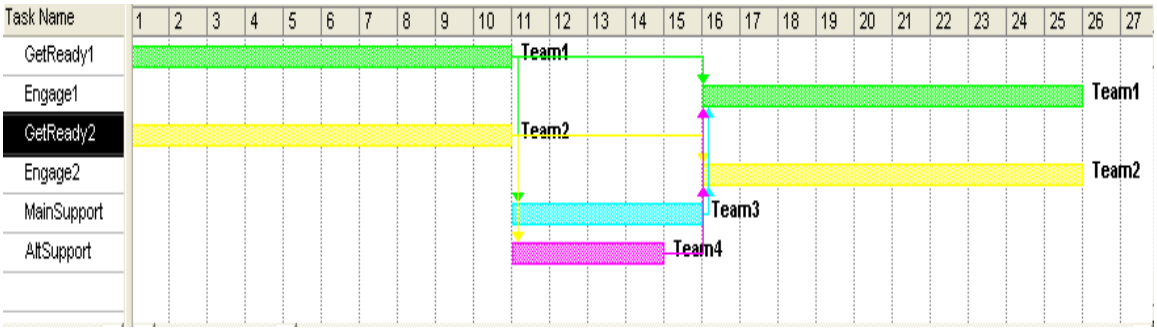


Figure 19. Gantt chart for Schedule3

Example 2: The example in Figure 20 is little more complex than Figure 16. It is rather a more practical example which demonstrates modeling in TAEMS [9]. It also has only one agent (may be the person interested in drinking coffee). The relevant details of the structure for Figure 20 are shown in Table 8, Table 9 and Table 10.

Table 8. Task Table Example 2

Task	**Subtasks**	**QAF**	**Release time**	**Deadline**
MakeCoffee	AcquireIngredients, HeatCoffee	Min	0	infinity
AcquireIngredients	FillWater, GetCoffee	Min	0	infinity
HeatCoffee	SetupBrewer, BrewCoffee	Sum	0	infinity
FillWater	GetHotWater,GetColdWater	Max	0	infinity
GetCoffee	AcquireGroundCoffeeBeans, UseInstantCoffee	Max	0	infinity
AcquireGroundCoffeeBeans	AcquireBeans, GrindBeans	Min	0	infinity
AcquireBeans	VisitStarbucks, UseFrozenBeans	Max	0	infinity

Table 9. Method Table Example 2

Method	Parents	Agent	Earliest Start	Deadline	Expected Duration	Expected Quality
GetHotWater	FillWater	Team1	0	100	4	6
GetColdWater	FillWater	Team1	0	100	2	6
VisitStarbucks	AcquireBeans	Team1	0	100	15	6
UseFrozenBeans	AcquireBeans	Team1	0	100	2	3
GrindBeans	AcquireGround CoffeeBeans	Team1	0	100	4	6
UseInstantCoffee	GetCoffee	Team1	0	100	1	1
SetupBrewer	HeatCoffee	Team1	0	100	4	6
BrewCoffee	HeatCoffee	Team1	0	100	6	7

Table 10. Enablement Table Example 2

Source Node	Target Node
AcquireBeans	GrindBeans
AcquireIngredients	SetupBrewer
SetupBrewer	BrewCoffee

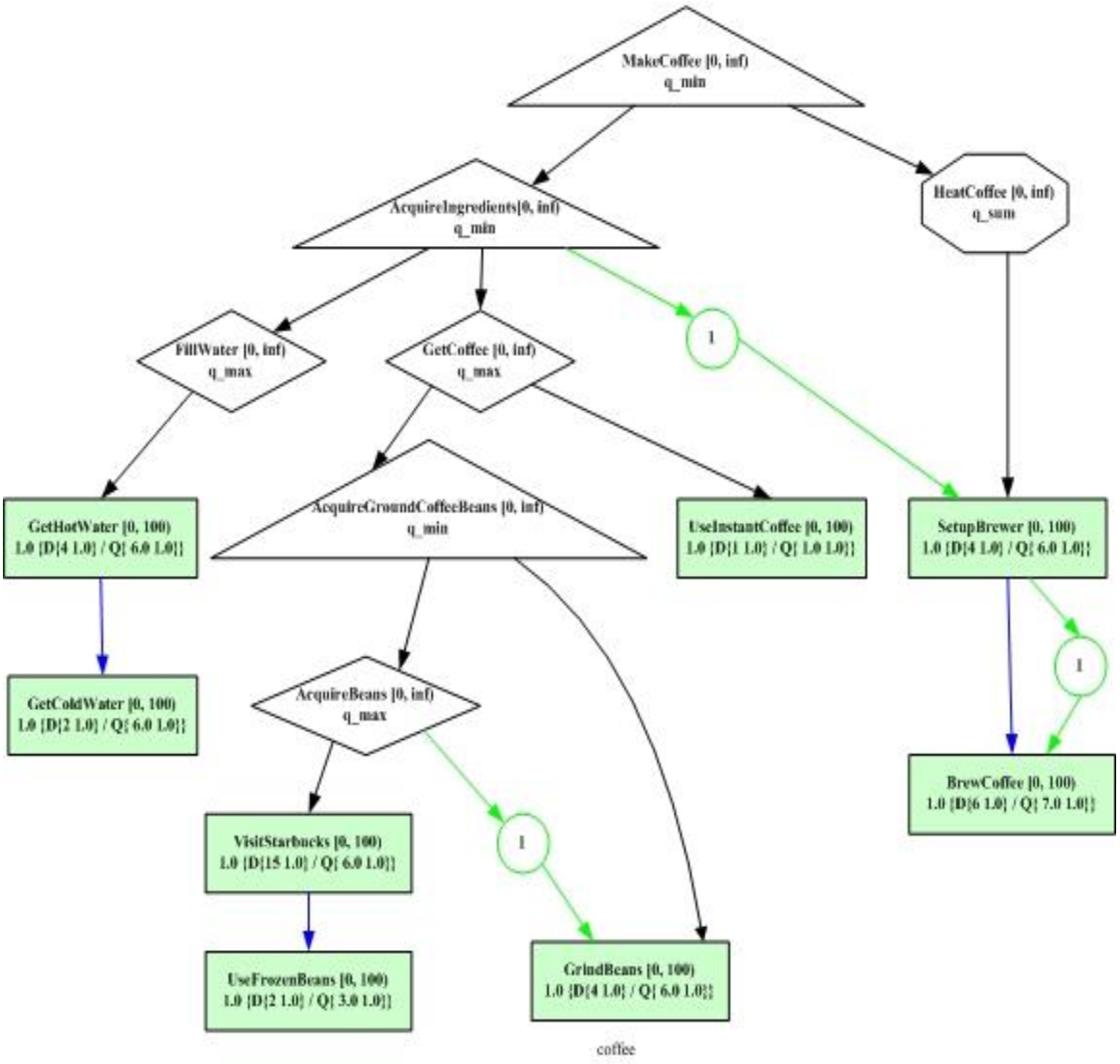


Figure 20 Coffee Making Example [9]

Plans generated from the above structure will outline the various ways in which an agent can succeed in making coffee. Then given the time it has at hand, there may be various schedules which determine the order and exact time of the tasks.

The planning constraints for the above task structure are written in the planning constraints ECLiPSe file given below:

```
:- lib(fd). plan(Coalist) :- Coalist = [UseFrozenBeans, VisitStarbucks,
SetupBrewer, GetColdWater, UseInstantCoffee, GrindBeans, BrewCoffee,
GetHotWater], Coalist :: 0..1, [AcquireGroundCoffeeBeans,
AcquireIngredients, HeatCoffee, GetCoffee, FillWater, MakeCoffee,
AcquireBeans] :: 0..1,MakeCoffee #= 1, MakeCoffee #<=>
(AcquireIngredients #/\ HeatCoffee), AcquireBeans #<=> (VisitStarbucks
#\/ UseFrozenBeans), AcquireGroundCoffeeBeans #<=> (AcquireBeans #/\
GrindBeans), HeatCoffee #<=> (SetupBrewer #\/ BrewCoffee), FillWater
#<=> (GetHotWater #\/ GetColdWater), GetCoffee #<=>
(AcquireGroundCoffeeBeans #\/ UseInstantCoffee), AcquireIngredients
#<=> (FillWater #/\ GetCoffee), SetupBrewer #=> AcquireIngredients,
GrindBeans #=> AcquireBeans, BrewCoffee #=> SetupBrewer, UseFrozenBeans
#=> (AcquireBeans #/\ AcquireGroundCoffeeBeans #/\ GetCoffee #/\
AcquireIngredients #/\ MakeCoffee), VisitStarbucks #=> (AcquireBeans
#/\ AcquireGroundCoffeeBeans #/\ GetCoffee #/\ AcquireIngredients #/\
MakeCoffee), SetupBrewer #=> (HeatCoffee #/\ MakeCoffee), GetColdWater
#=> (FillWater #/\ AcquireIngredients #/\ MakeCoffee), UseInstantCoffee
#=> (GetCoffee #/\ AcquireIngredients #/\ MakeCoffee), GrindBeans #=>
(AcquireGroundCoffeeBeans #/\ GetCoffee #/\ AcquireIngredients #/\
MakeCoffee), BrewCoffee #=> (HeatCoffee #/\ MakeCoffee), GetHotWater
#=> (FillWater #/\ AcquireIngredients #/\
MakeCoffee),labeling(Coalist). planall(L) :- findall(Coalist,
plan(Coalist), L).
```

This file is input to the ECLiPSe solver. The number of plans generated by the planner is 42. The scheduler output generates a schedule for all the 42 plans mainly because of lenient deadlines. Time Taken by scheduler to generate these 42 schedulers is approximately 93 ms. All the 42 schedules are shown in Appendix D. Gantt char representation of Schedule5 which is probably the preferred method of Coffee making is shown in Figure 21.

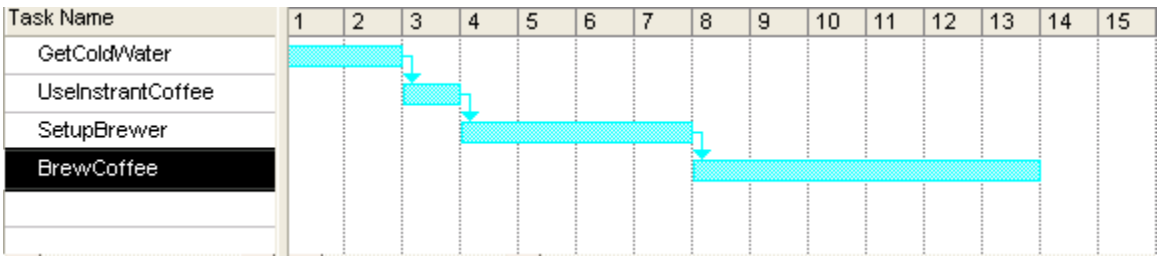


Figure 21. Gantt chart for schedule 5 example 2

Example 3: The examples above are simple instances of the task structure and hence have limited number of plans and schedules. But some structures are there having thousands of plans,

many of which can be scheduled. In such cases, total tree search runs out of processing memory. Even setting the local and global stack to maximum sizes of available RAM might result in out of memory exception. The general observation is that, since ECLiPSe and Java share the resources, ECLiPSe almost always succeeds in scheduling as many as 50000 plans. To avoid such a condition it is better to limit the search in some way and generate as many schedules as possible. Different search strategies can also be analyzed here. Consider the task structure in Appendix A. This is a fairly complex structure with innumerable number of plans. It has 30 methods, 16 Tasks, 4 agents and 5 enables relationship. If all the methods could be scheduled it would possibly have 230 plans. Using the normal labeling approach, the solver would result in an out-of-memory exception. Result of applying various limiting search strategies on this structure is compared below.

1) *Branch and bound search*: This makes use of the local variable 'backtracks' to keep track of the number of backtracks. Results are summarized in Table 11.

Table 11. BBS result summary

Backtrack Limit	**Number of Plans generated**	**Time Taken by planner (in ms)**	**Time Taken by scheduler (in ms)**
2000	3977	1062	300
5000	9949	6031	997
10000	20249	19906	120670
18000	35831	64215	216500

2) *Credit Search*: This incomplete search type controls the search and limits the solution found using credits assigned to each branch of the search tree. Results are summarized in Table 12. The observation is that credit search is not as good as BBS for planning problems though it is capable of generating more plans. Hence it is more memory efficient.

Table 12. Credit Search result summary

Credit Limit	Number of Plans generated	Time Taken by planner (in ms)	Time Taken by scheduler (in ms)
2350	3965	1750	670
5000	8592	6421	1330
10000	17593	20062	1609
11000	20000	23000	1890
27000	48443	188417	-

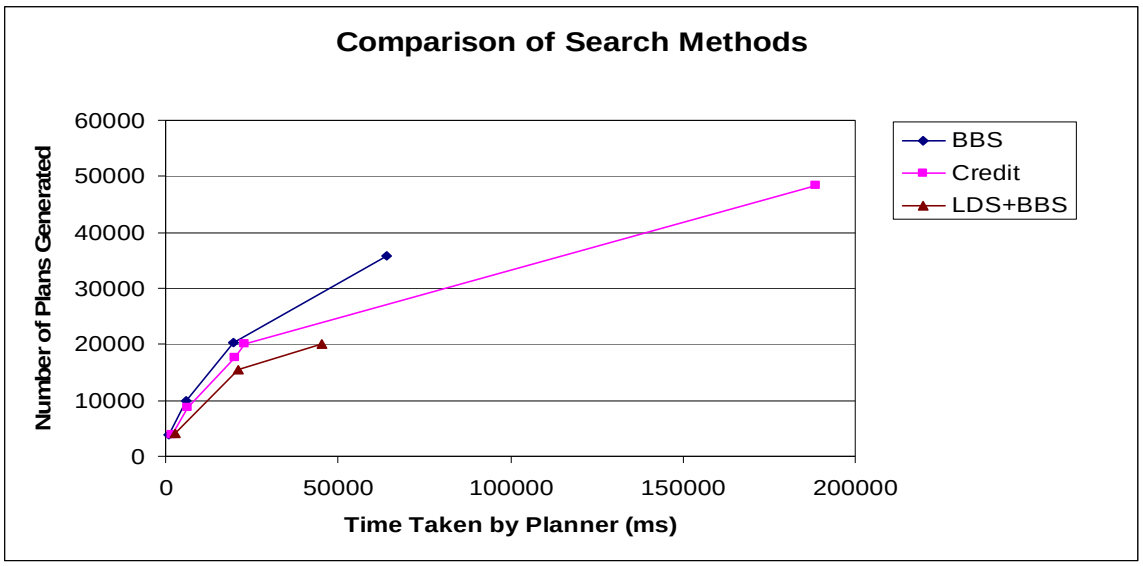
3) *LDS + BBS search*: This method combines the complete LDS, limited discrepancy search coupled with incomplete Bounded Backtrack search to efficiently find solutions. For search predicate : Coalist tent_set[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],bbs_dynamic_lds(Coalist,5000, D)

Table 13. BBS + LDS search summary

Backtrack Limit	Number of Plans generated	Number of Schedules generated	Time Taken by planner (in ms)	Time Taken by scheduler (in ms)
1300	4034	4034	2906	730
5000	15424	15424	21094	3644
6300	20092	20092	45295	4680

From the results in Table 11, Table 12 and Table 13, it can be observed that BBS is more time-efficient than the other two search processes. It generated more plans in relatively shorter time. This is the default search method in this thesis. Credit search successfully generated more number of plans than any other process under the same memory constraints. It can be concluded that credit search is better memory efficient than the others. BBS in conjunction with LDS has similar performance as credit search with respect to time but could not generate a huge number of plans. Hence it is seen that various search methods can be employed to compute as many plans as possible limited by hardware constraints (memory etc). The recommended search

method for this thesis for time efficiency is BBS search. For generating maximum number of plans credit search may be used. Shown below is a graphical comparison of the experimental search evaluation.



CHAPTER V

CONCLUSIONS AND FUTURE WORK

Conclusions

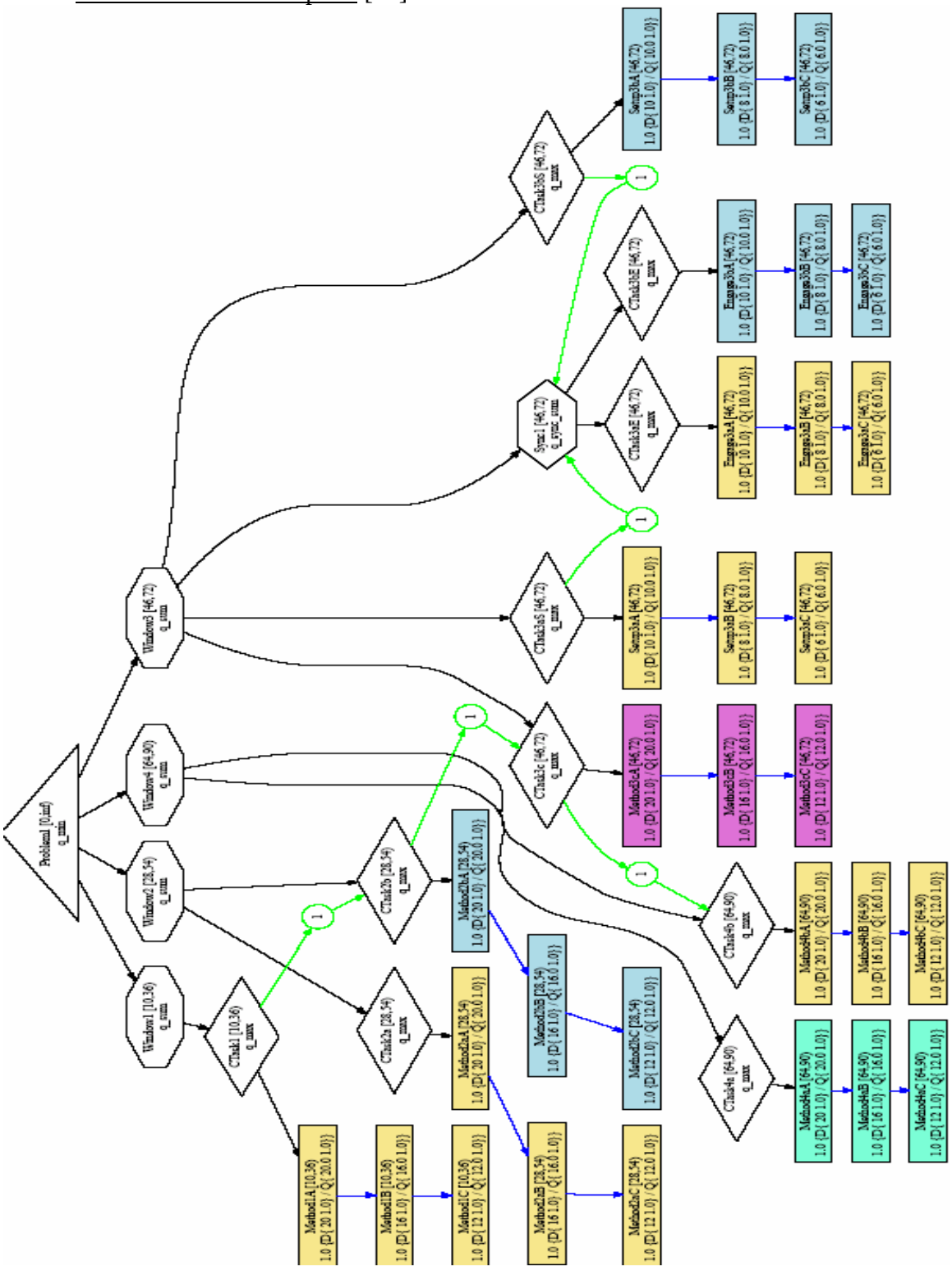
Planning and Scheduling is a broad and well-recognized research area in Artificial Intelligence. It deals with choosing a course of action to achieve a goal and computing a schedule for the actions constrained by certain resources. Many general purpose planning and scheduling approaches exist. The TAEMS planning and scheduling problem is a particular case, where the actions need to be scheduled to accomplish a root task in a graph like structure where there is uncertainty associated with the actions. This makes the problem an NP-hard problem. Various approaches can be taken to solve the problem including implementing efficient heuristics in a traditional programming language. The approach taken to solve the problem in this thesis is to convert the TAEMS scheduling problem into a Constraint Satisfaction Problem and use a Constraint Programming solver back-end to solve the base problem. The pre-processing of the task structure data is done in Java whereas the Constraint Programming Language used for backend solving is ECLiPSe. The solver output is parsed to generate various schedules possible for the given task structure, which is the course of actions and their associated start times. These actions when scheduled are expected to accumulate non-zero, positive quality at the TAEMS root task. This thesis shows the successful encoding of a complex planning scheduling problem into a Constraint Satisfaction Problem. Hence the built-in solver heuristics can be efficiently utilized to come up with schedules. The validity of the approach has been seen by creating schedules for several TAEMS task structures.

Future Work

This thesis has tried a constraint programming approach to solve the TAEMS planning, scheduling problem by converting the problem to a Constraint Satisfaction Problem. While it has satisfactorily generated schedules for many task structures, certain features can be extended to improve the solver. The solver can be extended to encode the optimality constraints, so that the schedules generated not only produce non-zero quality but high quality at task group. The probabilistic nature of the structure can also be handled by encoding the outcomes details as constraints. The framework can be changed to generate pessimistic schedules or more flexible schedules. Outcome analysis can be done to consider the probabilistic nature and encode the information in constraints. In addition, various heuristics can be developed using ECLiPSe to make the planning and scheduling process faster and more memory efficient.

APPENDIX A

Figure for CASE STUDY Example 3 [23]



APPENDIX B

ECLiPSe Bounded Backtrack Search Code modified for use in this thesis [6]

```
:- lib(fd).
:- lib(repair).
:- local variable(backtracks).
:- local variable(deep_fail).


bounded_backtrack_search(List,Limit) :-
    setval(backtracks,Limit),
    block(bbs_label(List),
          exceed_limit,
          (fail)
         ).

bbs_label([]).
bbs_label([Var|Vars]) :-
    limit_backtracks,
    indomain(Var),
    bbs_label(Vars).

limit_backtracks :-
        setval(deep_fail,false).
limit_backtracks :-
        getval(deep_fail,false),
        setval(deep_fail,true),
        decval(backtracks),
        (getval(backtracks,0) -> exit_block(exceed_limit) ; fail).
```

APPENDIX C

The Xml solver output for Example 1. It has all possible schedules for the structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<coaset taemsfile="Example1.ctaems">
    <coas label="CoA_0">
        <methodNames>GetReady1</methodNames>
        <methodNames>GetReady2</methodNames>
        <methodNames>AltSupport</methodNames>
        <methodNames>Engage1</methodNames>
        <methodNames>Engage2</methodNames>
        <schedule>
            <pairs methodname="GetReady1" time="0"/>
            <pairs methodname="GetReady2" time="0"/>
            <pairs methodname="AltSupport" time="10"/>
            <pairs methodname="Engage1" time="14"/>
            <pairs methodname="Engage2" time="14"/>
        </schedule>
    </coas>
    <coas label="CoA_1">
        <methodNames>GetReady1</methodNames>
        <methodNames>GetReady2</methodNames>
        <methodNames>MainSupport</methodNames>
        <methodNames>Engage1</methodNames>
        <methodNames>Engage2</methodNames>
        <schedule>
            <pairs methodname="GetReady1" time="0"/>
            <pairs methodname="GetReady2" time="0"/>
            <pairs methodname="MainSupport" time="10"/>
            <pairs methodname="Engage1" time="15"/>
            <pairs methodname="Engage2" time="15"/>
        </schedule>
    </coas>
    <coas label="CoA_2">
        <methodNames>GetReady1</methodNames>
        <methodNames>GetReady2</methodNames>
        <methodNames>MainSupport</methodNames>
        <methodNames>AltSupport</methodNames>
        <methodNames>Engage1</methodNames>
        <methodNames>Engage2</methodNames>
        <schedule>
            <pairs methodname="GetReady1" time="0"/>
            <pairs methodname="GetReady2" time="0"/>
            <pairs methodname="MainSupport" time="10"/>
            <pairs methodname="AltSupport" time="10"/>
            <pairs methodname="Engage1" time="15"/>
            <pairs methodname="Engage2" time="15"/>
        </schedule>
    </coas>
</coaset>
```

APPENDIX D

The Schedules generated for Example 2. It has 42 possible schedules for the structure.

```
Schedule1: GetHotWater: 0, SetupBrewer: 5, UseInstantCoffee: 4
Schedule2: BrewCoffee: 9, GetHotWater: 0, SetupBrewer: 5, UseInstantCoffee: 4
Schedule3: GetColdWater: 0, SetupBrewer: 3, UseInstantCoffee: 2
Schedule4: GetColdWater: 0, GetHotWater: 2, SetupBrewer: 7, UseInstantCoffee: 6
Schedule5: BrewCoffee: 7, GetColdWater: 0, SetupBrewer: 3, UseInstantCoffee: 2
Schedule6: BrewCoffee: 11, GetColdWater: 0, GetHotWater: 2, SetupBrewer: 7,
           UseInstantCoffee: 6
Schedule7: GetHotWater: 0, GrindBeans: 19, SetupBrewer: 23, VisitStarbucks: 4
Schedule8: BrewCoffee: 27, GetHotWater: 0, GrindBeans: 19, SetupBrewer: 23,
           VisitStarbucks: 4
Schedule9: GetHotWater: 0, GrindBeans: 19, SetupBrewer: 24, UseInstantCoffee: 23,
           VisitStarbucks: 4
Schedule10: BrewCoffee: 28, GetHotWater: 0, GrindBeans: 19, SetupBrewer: 24,
           UseInstantCoffee: 23, VisitStarbucks: 4
Schedule11: GetColdWater: 0, GrindBeans: 17, SetupBrewer: 21, VisitStarbucks: 2
Schedule12: GetColdWater: 0, GetHotWater: 2, GrindBeans: 21, SetupBrewer: 25,
           VisitStarbucks: 6
Schedule13: BrewCoffee: 25, GetColdWater: 0, GrindBeans: 17, SetupBrewer: 21,
           VisitStarbucks: 2
Schedule14: BrewCoffee: 29, GetColdWater: 0, GetHotWater: 2, GrindBeans: 21,
           SetupBrewer: 25, VisitStarbucks: 6
Schedule15: GetColdWater: 0, GrindBeans: 17, SetupBrewer: 22, UseInstantCoffee: 21,
           VisitStarbucks: 2
Schedule16: GetColdWater: 0, GetHotWater: 2, GrindBeans: 21, SetupBrewer: 26,
           UseInstantCoffee: 25, VisitStarbucks: 6
Schedule17: BrewCoffee: 26, GetColdWater: 0, GrindBeans: 17, SetupBrewer: 22,
           UseInstantCoffee: 21, VisitStarbucks: 2
Schedule18: BrewCoffee: 30, GetColdWater: 0, GetHotWater: 2, GrindBeans: 21,
           SetupBrewer: 26, UseInstantCoffee: 25, VisitStarbucks: 6
Schedule19: GetHotWater: 0, GrindBeans: 6, SetupBrewer: 10, UseFrozenBeans: 4
Schedule20: BrewCoffee: 14, GetHotWater: 0, GrindBeans: 6, SetupBrewer: 10,
           UseFrozenBeans: 4
Schedule21: GetHotWater: 0, GrindBeans: 6, SetupBrewer: 11, UseFrozenBeans: 4,
           UseInstantCoffee: 10
Schedule22: BrewCoffee: 15, GetHotWater: 0, GrindBeans: 6, SetupBrewer: 11,
           UseFrozenBeans: 4, UseInstantCoffee: 10
Schedule23: GetColdWater: 0, GrindBeans: 4, SetupBrewer: 8, UseFrozenBeans: 2
Schedule24: GetColdWater: 0, GetHotWater: 2, GrindBeans: 8, SetupBrewer: 12,
           UseFrozenBeans: 6
Schedule25: BrewCoffee: 12, GetColdWater: 0, GrindBeans: 4, SetupBrewer: 8,
           UseFrozenBeans: 2
Schedule26: BrewCoffee: 16, GetColdWater: 0, GetHotWater: 2, GrindBeans: 8,
           SetupBrewer: 12, UseFrozenBeans: 6
Schedule27: GetColdWater: 0, GrindBeans: 4, SetupBrewer: 9, UseFrozenBeans: 2,
           UseInstantCoffee: 8
Schedule28: GetColdWater: 0, GetHotWater: 2, GrindBeans: 8, SetupBrewer: 13,
           UseFrozenBeans: 6, UseInstantCoffee: 12
Schedule29: BrewCoffee: 13, GetColdWater: 0, GrindBeans: 4, SetupBrewer: 9,
           UseFrozenBeans: 2, UseInstantCoffee: 8
Schedule30: BrewCoffee: 17, GetColdWater: 0, GetHotWater: 2, GrindBeans: 8,
           SetupBrewer: 13, UseFrozenBeans: 6, UseInstantCoffee: 12
Schedule31: GetHotWater: 0, GrindBeans: 21, SetupBrewer: 25, UseFrozenBeans: 4,
           VisitStarbucks: 6
```

Schedule32: BrewCoffee: 29, GetHotWater: 0, GrindBeans: 21, SetupBrewer: 25, UseFrozenBeans: 4, VisitStarbucks: 6
Schedule33: GetHotWater: 0, GrindBeans: 21, SetupBrewer: 26, UseFrozenBeans: 4, UseInstantCoffee: 25, VisitStarbucks: 6
Schedule34: BrewCoffee: 30, GetHotWater: 0, GrindBeans: 21, SetupBrewer: 26, UseFrozenBeans: 4, UseInstantCoffee: 25, VisitStarbucks: 6
Schedule35: GetColdWater: 0, GrindBeans: 19, SetupBrewer: 23, UseFrozenBeans: 2, VisitStarbucks: 4
Schedule36: GetColdWater: 0, GetHotWater: 2, GrindBeans: 23, SetupBrewer: 27, UseFrozenBeans: 6, VisitStarbucks: 8
Schedule37: BrewCoffee: 27, GetColdWater: 0, GrindBeans: 19, SetupBrewer: 23, UseFrozenBeans: 2, VisitStarbucks: 4
Schedule38: BrewCoffee: 31, GetColdWater: 0, GetHotWater: 2, GrindBeans: 23, SetupBrewer: 27, UseFrozenBeans: 6, VisitStarbucks: 8
Schedule39: GetColdWater: 0, GrindBeans: 19, SetupBrewer: 24, UseFrozenBeans: 2, UseInstantCoffee: 23, VisitStarbucks: 4
Schedule40: GetColdWater: 0, GetHotWater: 2, GrindBeans: 23, SetupBrewer: 28, UseFrozenBeans: 6, UseInstantCoffee: 27, VisitStarbucks: 8
Schedule41: BrewCoffee: 28, GetColdWater: 0, GrindBeans: 19, SetupBrewer: 24, UseFrozenBeans: 2, UseInstantCoffee: 23, VisitStarbucks: 4
Schedule42: BrewCoffee: 32, GetColdWater: 0, GetHotWater: 2, GrindBeans: 23, SetupBrewer: 28, UseFrozenBeans: 6, UseInstantCoffee: 27, VisitStarbucks: 8

REFERENCES

[1] P. Baptiste, C. Pape and W. Nuijten, Constraint Based Scheduling: Applying Constraint Programming to Scheduling Problems, Kluwer Academic Publishers Boston 2001.

[2] K. Marriott and P.J. Stuckey, Programming with Constraints: an Introduction, MIT Press, Boston, 1998.

[3] R. Bartak, *Constraint-Based Scheduling: An introduction for Newcomers*, Technical Report TR 2002/2, Department of Theoretical Computer Science and Mathematical Logic, Charles University, 2002.

[4] M. Ginsberg and W. Harvey, Limited Discrepancy Search, Proceedings of the Fourteenth International Joint Conference of Artificial Intelligence (IJCAI-95); Vol. 1, pp 607-613, 1995.

[5] The Mozart Programming System, www.mozart-oz.org.

[6] The ECLiPSe Constraint Logic Programming System, http://eclipse.crosscoreop.com/eclipse/

[7] M. Yokoo, *Distributed Constraint Satisfaction: foundations of cooperation in multi-agent systems*, Springer 2001.

[8] M. Ghallab, D. Nau and P. Traverso, Automated Planning: theory and Practice, Morgan Kauffman Publishers, 2004

[9] B. Horling, V. Lesser, R. Vincent, T. Wagner, A. Raja, S. Zhang, K. Decker and A. Garvey, The TAEMS White Paper, January 1999.

[10] B. Hayes-Roth and F. Hayes-Roth, A Cognitive Model of Planning, International Joint Conference on Artificial Intelligence, 1979.

[11] M. A. Peot, D. E. Smith, Conditional Non-Linear Planning, *Proceedings of the first International Conference on AI Planning System*, College Park, Maryland, 1992.

[12] R. Vincent, B. Horling and V. Lesser, Experiences in Simulating Multi-Agent Systems Using TAEMS, *The Fourth International Conference on MultiAgent Systems (ICMAS 2000)*, AAAI, July 2000.

[13] C. A. Knoblock, An analysis of ABSTRIPS, Morgan Kaufmann Publishers Inc. San Francisco, CA, 1992.

[14] D. Jensen, M. Atighetchi, R. Vincent and V. Lesser, Learning Quantitative Knowledge for Multiagent Coordination, *16th National Conference on Artificial Intelligence (AAAI-99)*, American Association for Artificial Intelligence, pp. 24-31, August 1999.

[15] T. A. Wagner, A. J. Garvey and V. R. Lesser, Criteria Directed Task Scheduling, *Journal for Approximate Reasoning (Special Issue on Scheduling)*, Volume 19, Elsevier Science Inc., pp. 91-118. January 1998. [A version is also available as UMass Computer Science Technical Report 1997-59.]

[16] K. Decker, TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms in *Foundations of Distributed Artificial Intelligence, Chapter 16*, G. O'Hare and N. Jennings (eds.), Wiley Inter-Science, pp. 429-448, January 1996.

[17] V. Lesser, M. Atighetchi, B. Benyo, B. Horling, A. Raja, R. Vincent, T. Wagner, X. Ping and S. XQ Zhang, The Intelligent Home Testbed, *Proceedings of the Autonomy Control Software Workshop (Autonomous Agent Workshop)*, January 1999.

[18] The CHIP System (white Paper), http://www.cosytec.com/production_scheduling/chip/pdf/the_chip_system.pdf.

[19] C. Pape, Implementing of Resource Constraints in ILOG SCHEDULE: A library for the Development of Constraint-based Scheduling Systems, in Intelligent Systems Engineering, vol3, 1994.

[20] C. vanBuskirk, B. Dawant, G. Karsai, J. Sprinkle, G. Szokoli and K. Suwanmongkol, *Computer-Aided Aircraft Maintenance Scheduling*, ISIS Technical Report, 2002.

[21] TAEMS Java API http://mas.cs.umass.edu/research/mass/api/taems/.

[22] Choco Solver, http://choco.sourceforge.net/.

[23] K. Decker and A. Garvey, Scenario Generation, *Automatically generating Domain independent Coordination Scenarios.* [Draft Version].

[24] M. Boddy, B. Horling, J. Phelps, R. Goldman, R. Vincent, C. Long and B. Kohout, C_TAEMS Language Specification, Version 1.06.

[25] K. Decker, Environment Centered Analysis and Design of Coordination Mechanisms, PhD Dissertation, Department of Computer Science, UMass, Amherst.

[26] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Chapters 11-13.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stien, Introduction to Algorithms, MIT Press, Cambridge, 2001.

[28] V. Lesser, B. Horling, F. Klassner, A. Raja, T. Wagner, S. XQ. Zhang, "BIG: An Agent for Resource-Bounded Information Gathering and Decision Making", *Artificial Intelligence, Special Issue on Internet Applications*, May-2000, vol 118, issue 1-2, pages 197-244.