NOTICE

Chapter 4 Solving Supply Chain Scheduling Problems with ODO

This chapter presents our approach to solving the supply chain scheduling problem. We begin by briefly describing the problem, followed by a discussion of constraint-based problem solving and constrained heuristic search. Also defined are the initial version of ODO, and our extensions to that model. We encapsulate the declaration of our search mechanism and accompanying algorithm(s) within one structure, called a problem solving *policy*. A *meta*-policy defines the control structure of the policy hierarchy, whereas an *atomic*-policy embodies a particular scheduling heuristic. This chapter also discusses the specification of our policies in PODL (**O**DO **P**roblem **D**escription **L**anguage).

4.1 The Supply Chain Scheduling Problem

The typical supply chain scheduling problem is defined by:

1. A set of generic **process-plans** (or, activity templates), one for each product. Each process plan is composed of a number of **activities**, all of which are completely defined within the scope of the process plan. The execution of a single instance of any process plan results in the procurement of a fixed, pre-determined quantity of the corresponding product.

   Each process-plan is a conjunctive / disjunctive network of activities. In other words, each process-plan may have one or more **OR** and **XOR** decisions points within its network. In the case of an XOR decision point, only one process path is to be selected from the alternatives. Thus each activity has a parameter called percentage-demand, which specifies the demand that activity satisfies, as a percentage of the total demand being satisfied by its parent process plan.

2. A set of **orders** for products, with each order parameterized by one product, required quantity, order due-date, release-date, latest-finish date, and required code-age. Release date specifies the earliest date the order can start being processed; latest end date specifies the time by which an order must be fulfilled. These two time points set the earliest and latest time bounds on the activities corresponding to an order. The code-age specifies the minimum remaining shelf-life for the product to be allocated to an order.

3. Each activity is further defined by start-time, duration, and end-time intervals. It is linked to one or more other activities via precedence constraints. An activity also uses / consumes / produces **resources**. Each resource-request specifies a specific resource or a pool of resources, and the quantity required of that resource / set. Each activity which produces a resource is also parameterized by a rate of production.

4. Resources are defined as objects, which play certain **roles** in activities. We model three classes of roles: mechanism, material, and container role. Mechanism role refers to roles which are a mechanism in the execution of activities, like machines, operators, factories, and so on. Material role refers to roles like raw material and inventory (work-in-process, finished products). Container role is associated with warehouses and storage areas.

   Each of these role-classes has its own definition of **capacity**, and we specify an maximum capacity for all resources. Mechanism role is linked to simultaneity capacity, which defines how many activities can a resource support simultaneously. Material role is linked to amount or discrete quantity in number of units, whereas container role corresponds to storage capacity.

   Resources can also be aggregated into sets, which is especially useful to model inventory. For example, an inventory of widgets can be composed of two sets, each having a set-size of 200 pencils. An activity which require 10 pencils can take them from either set, leaving it with 190 pencils.

5. We also model **shelf-life** of inventory; shelf-life specifies the number of time units from the time of manufacture that inventory will spoil. Inventory which is spoilt can no longer be used for its intended purpose, and thus represents a high cost.

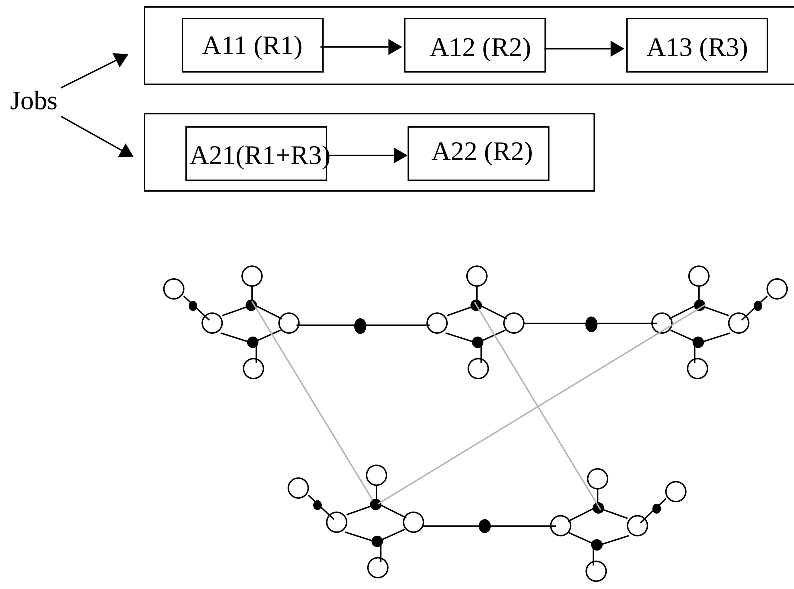
4.2 Constraint-Based Problem Solving

One of the general techniques for representing and solving constrained problems is called *Constraint Satisfaction*. According to [Mackworth 86] , the problem is represented by a *constraint graph* (Figure 9, "Constraint Graph," on page 63) in this approach. The nodes represent variables, and the arcs connecting these nodes represent the constraints between them. Thus, solving a constraint satisfaction problem (CSP) is tantamount to assigning values to variables such that all constraints are satisfied [Davis 94] .

There are many advantages of selecting a constraint based representation for a problem. First, many problems naturally lend themselves to this representation; they are easily expressed in terms of variables and constraints. Secondly, it is easy to create variations of a problem by adding / deleting variables and / or constraints. This is important in dynamic domains where it may be necessary to solve several versions of a problem [Davis 94] . It is also an important property to solve problems in which not all variables and constraints are known prior to problem solving.

However, the most important reason to choose this representation would be to exploit its constraint graph structure during problem solving. The constraint graph enables a mechanism called *consistency enforcement*: when a variable's value / domain is modified, the change can *propagate* through the graph and alter the actual / possible values of other variables [Davis 94] . Many consistency enforcement and search techniques have been developed for CSP's [Gaschnig 77] [Haralick 80] [Bitner 75] . The constraint based problem structure can also be utilized by domain specific *heuristics* during search or propagation [Davis 94] . The following figure presents a small problem, and its corresponding constraint graph.

Figure 9. Constraint Graph



Empty circles are variables. Solid circles are constraints connecting these variables. Solid lines are temporal constraints; dotted lines are resource constraints.

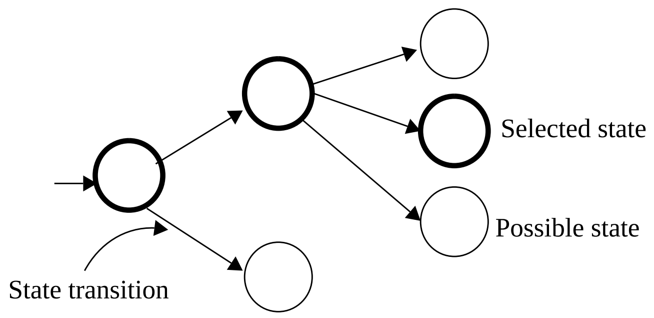
Problem solving is performed by sequentially selecting a variable and assigning a value to that variable. From the perspective of heuristic search, the initial state of a problem contains all variables, their associated domains, and the constraints acting over these variables. The heuristic operators select a variable at each step, and a value to assign to it. The evaluation function may be composed of some constraints, evaluation metrics, and an objective function [Fox 89] .

There exist many incremental search heuristics which focus on generic overall variable and value selection strategies that will lead to an efficient solution [Davis 94] . Two most commonly encountered examples are *constructive* (backtracking) and *repair-based* search. In constructive search, each new assignment that is made is consistent with all previous assignments. Thus, at any given step, the partial solution is consistent. Backtracking occurs when there can be no complete solution with current assignments. In this situation, one or more previously made assignments are revoked. In repair based search, the initial state has all variables assigned, even if these assignments are inconsistent. Search then involves repeatedly changing assignments on variables, or, making repairs. The goal of this process is to reduce the number of inconsistencies. As in constructive search where we can backtrack to a previous state, repair based methods can also revert to a previous state.

4.2.1 Search as Commitment Transformation

Search is sometimes viewed as the traversal of problem-solving states via state transition operators [Davis 94] . Each search transition transforms one problem solving state into another, as shown below. Search is terminated once an acceptable state is reached. Some systems explicitly represent the search space in this manner. The "belief" is that each decision transforms the problem state to one closer to the desired state.

Figure 10. Search as Commitment Transformation



We can consider a typical backtracking search procedure as a transformational search. The actual transformations being performed are assignment of values to variables as new assertions are made, and retracting previous assignments during backtracking. Upon each assignment, consistency is enforced through propagation, which alters the values or domains of other variables. Thus, consistency enforcement is another way of making new assertions. The search process may also involve posting and retracting of constraints instead of variable/value assignments. Both constraint posting and variable/value assignments create new states that post restrictions of the domains of variables when going forward, and release these restrictions when backtracking.

This leads us to suggest that general search consists of making transformations that create new *commitments*, and when backtracking occurs, more transformations *release* these commitments [Davis 94] . Repair-based methods follow these commit-release transformations as well. Each new assignment of a value to a variable release the previous commitment, and makes a new one. More commitments are made in case of propagation.

4.3 ODO: Version 1

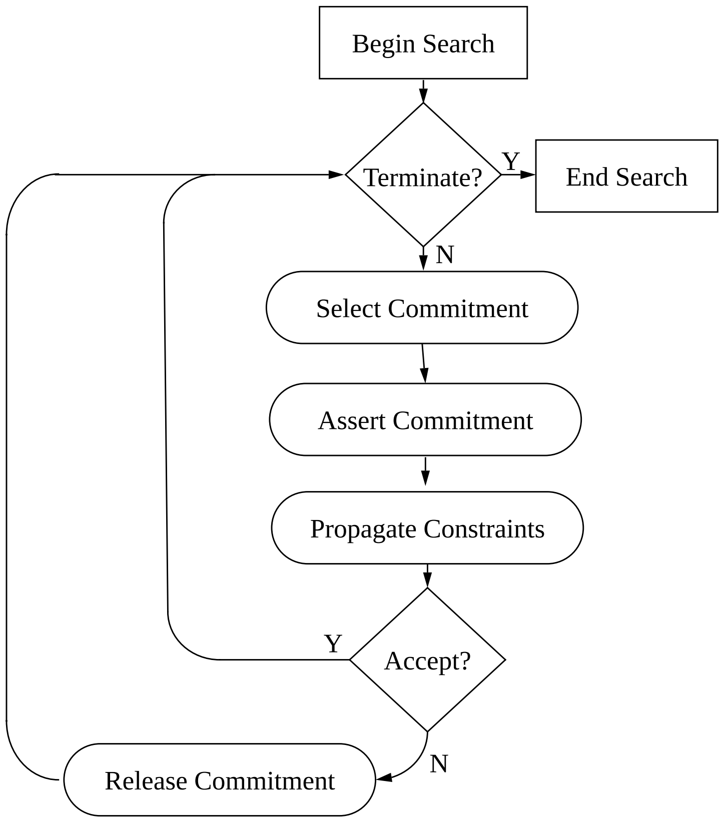
This section presents the problem solving model of ODO: Version 1, which provided the initial basis for this work. ODO is founded on the belief that there is an opportunity to discover some of the associations between problem structure and heuristics performance in constraint-based scheduling [Davis 94] . The approach is to view the modeling and solution of a constraint based scheduling problem from a unified model that combines common components and isolates essential differences.

4.3.1 The Unified Model

By employing a constraint based representation, all problems are formulated using a constraint graph. It has been demonstrated that graph property measurements can characterize the heuristic search performance [Davis 94] [Fox 87] . [Fox 87] introduced *textures* as properties of a constraint graph that can guide heuristic decision making. This concept clearly distinguishes between the constraint graph information (texture measurement) and heuristic decision making. ODO thus exploits past research into the relationships between graph properties and heuristic performance.

Many constraint-based schedulers perform incremental search of the problem space within the bounds of a generic “template.” At each step, a modification of the problem space is chosen. In general, this modification asserts a *commitment* or retracts a previous one. After the modification has been asserted, the resulting changes are propagated through the problem space via constraints. The resulting state is accepted or retracted based on certain criteria. This incremental search is performed until the search termination criteria are met. This basic search loop is illustrated in the following figure.

Figure 11. General Search Loop



In fact, the above loop is also a generic representation of many heuristic search approaches. Each heuristic is characterized by what type of commitments are allowed, how commitments are chosen, how much constraint propagation is performed, how to release a commitment, and what the acceptance and termination criteria are [Davis 94] ? ODO defines a **policy** as the exact specification of how each step within this loop is performed. Texture measures guide the decisions made at these steps.

ODO is a constraint-based scheduling system that implements the above unified model. Search is performed using the above loop, over a constraint graph representation. The problem to be solved and the solution strategy (i.e., the search heuristic) are both input via an input language, which makes ODO an "interpreter", since heuristics are created at run time. Many well known scheduling heuristics like Min-Conflicts and MicroBOSS have been reconstructed within ODO.

4.3.2 ODO's Problem Solving[1]

Problem solving within ODO involves the declaration of all policy parameters followed by the initiation of ODO's search mechanism. Extending the unified model of ODO within the context of search using variable and value selection, a complete policy specification involves:

- How to select candidate variables(s)?
- How to select / generate values for these variables?
- How to evaluate variable / value assignments, i.e., how to evaluate potential commitments?
- How to select one variable / value assignment?
- How to perform propagation?
- How to evaluate and accept / reject resulting state?
- When to terminate the search?

A library of texture measures is provided within ODO to guide the performance of each policy step. Version II follows the same general search loop as described above so we will briefly review each of these policy steps and the associated options. Further, we will also present ODO's reconstruction of MicroBOSS [Sadeh 91] , which has also influenced the design of our heuristics.

ODO implements a 2-tiered policy hierarchy: a parent **meta**-policy and its children **atomic**-policies. Each atomic policy represents an actual problem solving policy. A meta policy is the overall mechanism for executing search on a given problem, using a set of atomic policies. The following are the procedures performed in the execution of an atomic policy. Each procedure's input is a list of **filter functions**. Each filter function performs a texture measure of the constraint graph. It takes a list or variables or variable / value pairs as input, and returns a subset of that list as output. Thus, it *filters* down a list of variables.

1. See Appendix X for details on each policy step and filter functions.

4.3.2.1 Variable and Value Selection

Four procedures are sequentially executed:

1. **Variable Selection**: filters an input list of variables down according to the variable selection filters.
2. **Value Generation**: accepts the above filtered list and generates a value for each variable in that list.
3. **Score Var/Val Pairs**: each var / val pair is scored on that basis of a scoring function.
4. **Select Var/Val Pair**: *one* var / val pair out of all scored pairs is selected using a selection filter.

4.3.2.2 Constraint Propagation

Propagation is performed once a variable and a value for it are selected. Both resource and temporal propagation are performed. The propagate procedure accepts as input the selected variable / value pair and a list of propagation filter functions. These functions are executed in sequence on the variable / value pair. A propagation function may enforce consistency on either actual or possible values for a variable. Temporal and resource consistency are performed independently, as is the common practice in scheduling. Full temporal consistency can be achieved more efficiently than resource consistency. The resource propagation functions thus achieve partial resource arc consistency.

4.3.2.3 Accept Criteria

After variable / value assertion and propagation, the resulting state is evaluated against the acceptance criteria in order to determine whether to accept the new state or to restore the previous state.

4.3.2.4 Backtrack

ODO implements only chronological backtracking mechanism which means that no declaration is required.

4.3.2.5 Search Termination

Prior to starting a new iteration, the current state is evaluated to determine whether the search termination criteria have been met. Once the criteria are met, search can be terminated. The termination expression allows for all logical relations and predicates, arithmetical negation, and some problem state measurement functions.

4.3.2.6 Cost Function

It is convenient at many steps during the search process to base decisions on the *cost* of the current state. Cost functions evaluate the current state and return an integer value. For instance, the termination criteria would often compute the cost of the current state as part of termination decision.

4.3.3 ODO's Implementation of MicroBOSS

ODO's implementation of MicroBOSS is based upon the version presented in [Sadeh 91] . Given ODO's capabilities as a scheduler interpreter, it is possible to reconstruct many heuristics within the unified problem model. MicroBOSS is of particular interest to us since it demonstrates the successful use of **demand-based textures** in solving job-shop scheduling problems. We have adapted and extended this concept to form the essential component of our heuristics.

MicroBOSS relies upon a constructive approach to generate a satisfying schedule[1]. The basic idea is to find the most constrained variable at each step and assign it the least constraining value. This means that the most critical decisions are made early on. This approach has been shown to minimize backtracking and thrashing. The search is *micro-opportunistic* in that it recalculates all decision making parameters at each step, thus dynamically tracking emerging opportunities in the search space.

For each unscheduled activity and the resource(s) it requires, MicroBOSS generates a *demand profile*, which is the probability that the activity will require a particular resource at a given time. Demand profiles associated with each resource are algebraically added to give the

1. Sadeh [Sadeh 91] presents MicroBOSS as a constrained optimizer scheduler. However, the version reconstructed within ODO employs only its variable/value selection heuristics.

aggregate demand profile of that resource. The resource with the maximum aggregate demand is the *most-contended-for* resource. The activity contributing the most to this aggregate demand is the most constraining activity, or, is most *reliant* on this resource. This is the activity selected for being scheduled at this step; the heuristic is called ORR (Operations Resource Reliance).

The next step is to select a value for this activity (variable), one that is least constraining. This is done by determining that which value results in a partial schedule with high *survivability* and *compatibility*. Survivability is a measure of that particular schedule surviving future assignments. Compatibility gives a measure of how many schedules would be compatible with this one. This heuristic is called FSS (Filtered Survivable Schedule). Both ORR and FSS are implemented as filter functions within the library of ODO. The results of this reconstruction, while not identical to Sadeh's MicroBOSS, are quite close and consistent with other efforts to duplicate MicroBOSS [Davis 94] .

4.4 Extensions to the Initial Model

In chapter three we presented the problem representation of ODO version I, and our extensions to and deviations from that model. Similarly in this section, we present our extensions to and variations from the initial problem solving model. We follow ODO's proposition of a unified problem solving model (see Figure 11, "General Search Loop," on page 66). This version also performs search as a commitment based transformation. However, in terms of decision making heuristics, many of our ideas are founded in demand-based textures, proposed by Sadeh and Fox [Sadeh 91] [Fox 87] . Some of the proposed extension to Sadeh's demand-based textures are along the lines of those proposed in KBLPS [Saks 93] [Saks 93] [Saks 92] . In the remainder of this section, we describe in detail our problem solving model.

4.4.1 What is a Commitment?

At each problem solving iteration, a commitment is made or retracted which transforms the current search state into another. The definition of a commitment is completely domain dependent. For example, we could assign a value to a variable, create new variables and/or constraints, and so on. In the first version of ODO, making a commitment was assigning a value to the start-time variable of an activity, or reducing its domain. However, as our problem is much more involved than a basic job-shop scheduling problem, we also have a more complicated notion of what a commitment is? There are three main, distinct types of commitments we are required to make in order to arrive at a solution. These are:

1. Decide which activities will satisfy how much demand, in a disjunctive process plan? This can also be looked at as which activities to execute?

2. Assign specific resources to activities, selecting from the set of alternatives available.

3. Sequence the schedule-able activities; instead of assigning start times to activities, we create new temporal constraints between a pair of activities, which has the effect of reducing their domains.

We view these three decisions as different types or “classes” of commitments we have to make, in order to solve our problem. Thus, we encapsulate the decision making heuristic(s) for each of these commitments in a separate and distinct set of atomic-policies, all of which are in turn are modeled as sub-policies of one meta-policy, in order to form a complete algorithm.

4.4.2 Texture Measures

In order to make “good” commitments, we need to address two points. First, we should be able to discern a “good” decision from a “poor” one. In other words, we need a mechanism to measure the “goodness” of a decision. Second, we require information on the state of the problem, vis-a-vis our objective(s), which we can then use as an input to our “goodness” measurement mechanism. Thus, we both need to *know* the search state and to be able to *process* that knowledge in order to make commitment related decisions.

We obtain this knowledge through the **texture** measures we define on the constraint graph representation of the problem. Different heuristics use these measures differently, in order to make “goodness” calculations, and we present these in Section 1.5.1 on page 63. The two fundamental texture measures which are employed in our work are:

- **Individual demand** of activities on resources (a measure of reservation *reliance*).
- **Aggregate demand** of all activities on resources (a measure of resource *contention*).

As we have pointed out earlier, the use of demand-based or contention-reliance based metrics has recently been shown to produce very strong performance in solving job-shop scheduling problems. We especially draw on the work of [Sadeh 91] and extend it towards solving the supply chain scheduling problem.

A *reservation* is defined as the assignment of a specific activity to a specific resource at a specific time. The reservation reliance of an activity on a resource is a function of the number of alternative reservations available for that activity, in that search state. It is obvious that an activity with a small set of possible reservations will rely heavily (pose high demand) on each of those reservations, as compared to an activity with a large set.

Let us first consider the most basic case[1]: each activity requires a single, specific resource of unit capacity. Thus, no two activities can overlap on the same resource. Further, in a given search state, each activity has a set of possible (say, start time) reservations. In that case, assuming no biases, each of these reservations is equally likely. Thus for each possible assignment ρ for an activity A_i, we have the following probability $\{\sigma_i(\rho)\}$ of being selected (allocated):

$$\sigma_i(\rho) = 1/nA_i \qquad \textbf{(EQ 3)}$$

where nA_i = number of possible reservations for activity A_i in this state. Clearly, if nA_i is greater, $\sigma_i(\rho)$ is smaller, i.e., more possible reservations means lesser reliance on any one of them. This probability is used to compute the individual demand *profile* of an activity on a resource. The demand of activity A_i on resource R_j at time τ is denoted by $D_i(R_j,\tau)$. This demand is computed by adding the probabilities $\sigma_i(\rho)$ of all reservations ρ of activity A_i that require resource R_j at time τ. Mathematically,

$$D_i(R_j, t) = \sum_{t-du_i}^{t} \sigma_i(\tau) \qquad \textbf{(EQ 4)}$$

1. For all formulae & discussion related to this basic model, also refer to [Sadeh 91] .

where du_i is the duration of activity A_i. The plot of demand vs. time is called the *individual demand* **profile** of that activity on that resource. Thus, we obtain our first texture measure, individual demand of activities on resources.

From this point, it is quite straightforward to obtain the other texture measure, the contention on resources. *Aggregate* demand on resources gives us a measure of resource contention. The aggregate demand on a resource is given by the *algebraic* sum of all the individual demand profiles related to that resource. This is expressed as:

$$D^{aggr}_{R_j}(t) = \sum_i D_i(R_j, t) \qquad \textbf{(EQ 5)}$$

Figure 12, “Demand Profiles,” on page 75 shows examples of both individual and aggregate demand profiles. There are three[1] extensions to the base case discussed above:

- activities have *non-unit requirements.*
- activities have a set of *alternative* resources to choose one from.
- resources have *non-unit capacity.*

Let us consider these on by one. When activities have non-unit requirements, each activity may require a different amount of a given resource. This implies that even for two activities which are identical in all respects except the amount they require, the demands should be different. Further, it is obvious that an activity which requires a greater amount, poses a higher demand. However, the probabilities used in computing the demand profile are unchanged (EQ 3)(EQ 4). Thus, the following equation gives the individual demand in case of non-unit requirements.

$$D_i'(R_j, t) = D_i(R_j, t) \times amt(A_i) \qquad \textbf{(EQ 6)}$$

where $D'_i(R_j,\tau)$ is the modified demand, $D_i(R_j,\tau)$ is the base demand computed from initial probabilities, and, *amt*(A_i) is the amount (units) of resource R_j required by activity A_i. This is also consistent with the approach for inventory request allocation to supply routes presented in KBLPS [Saks 93] [Saks 93] [Saks 92] .

1. The situation that an activity requires multiple resources conjunctively (simultaneously) does **not** change its reliance on any *single* resource.

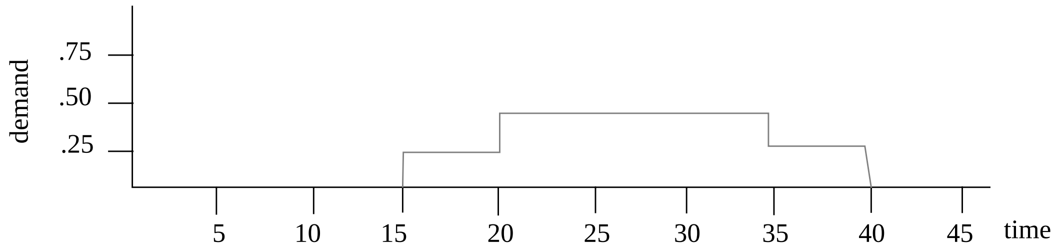
For the second extension, that activities have a set of alternatives to choose one resource from, we follow the same reasoning as in deriving the probabilities. It is logical to say that if an activity has a large set of alternative resources, it relies on any one of them less. Conversely, if it has a small set to choose from, it relies heavily on each alternative. Thus, the demand of an activity on a single resource is inversely proportional to the number of alternative resource available for that activity. The following equation gives us the modified demand in case of alternative resources:

$$D_i'(R_j, t) = D_i(R_j, t) \times \frac{1}{alt(A_i)} \qquad \text{(EQ 7)}$$

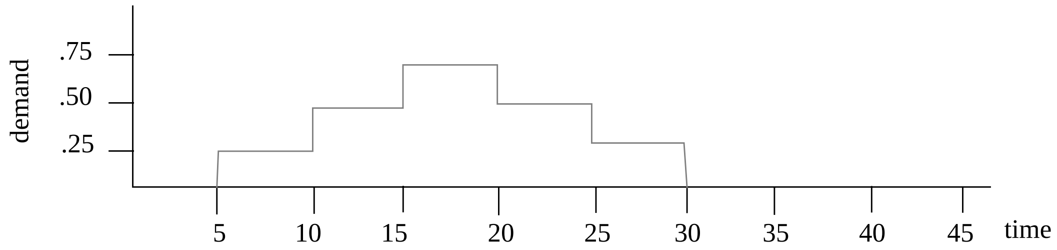
Again, the above extension is consistent with KBLPS's reasoning on reliance of an inventory request on a supply node. It is also important to re-emphasize that there are no preferences among alternatives. Combining the above two extensions, the individual demand of an activity on a specific resource is given by:

$$D_i''(R_j, t) = D_i(R_j, t) \times \frac{1}{alt(A_i)} \times amt(A_i) \qquad \text{(EQ 8)}$$

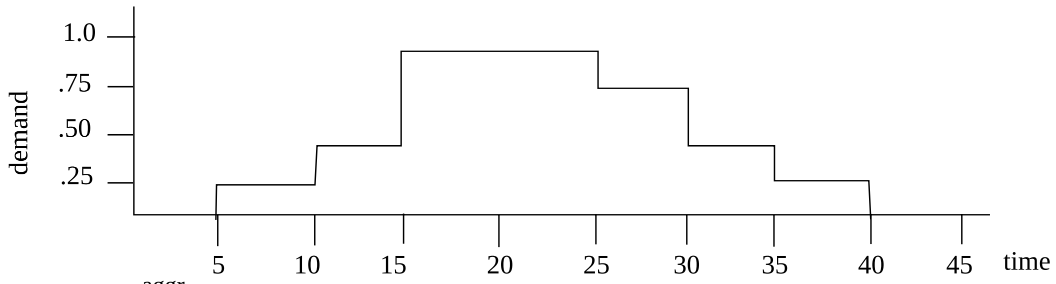
Figure 12. Demand Profiles



$D_1(R_1,\tau)$: Individual Demand of activity A_1 on resource R_1.



$D_2(R_1,\tau)$: Individual Demand of activity A_2 on resource R_1.



$D^{aggr}{}_{R1}(\tau)$: Aggregate Demand on resource R_1.

The third case is of resources having different, non-unit capacities. It is evident that all other factors being equal, a resource having a smaller capacity will be more heavily contended for, since it has less to give. That is, an increase in capacity of a resource lowers the contention on it. Therefore, our measure of contention, aggregate demand on a resource, should also be inversely proportional to resource capacity. The modification is given by:

$$D^{aggr'}_{R_j}(t) = D^{aggr}_{R_j}(t) \times \frac{1}{capacity} \quad \textbf{(EQ 9)}$$

This in fact, normalizes the aggregate demand profile on each resource. We are also using another ‘normalization’ of the aggregate demand curve; one which compares *real* demand with *real* capacity. We are using this criteria specifically to terminate search, as it gives an accurate measure of “real” contention on the resource. The normalization function is:

$$D_{R_j}^{aggr'}(t) = max\{RD_{R_j}^{aggr} - Capacity, 0\} \quad \textbf{(EQ 10)}$$

where RD represents "real demand" (as opposed to probabilistic). This measure ensures that only those points in time where the resource is actually violated are reflected in the real-demand curve, and so this can effectively be used as a termination criteria for one of our policies the purpose of which is to allocate all activities to various resources such that all resources are feasible (Section 4.5.6 on page 95). The resource are considered feasible if their capacity at all points in time equals or exceeds demand for them at that time. The profile of actual demand (as opposed to probabilistic demand) with time gives us that information. Thus real demand curves on resources (EQ 10) are used to terminate that policy.

Finally, we should also consider the case of *disjunctive process plans*; specifically, the **percentage-demand** associated with activities and how it may affect demand-profile calculations. The percentage-demand parameter represents the demand for the end-product being satisfied through this activity, as a percentage of the demand being satisfied by the parent process-plan. It is quite rational to state that activities with high percentage-demands are more critical within a process-plan, or, require resources more urgently. In other words, reliance is proportional to percentage-demand. This modifies our calculation as:

$$D_i'(R_j, t) = D_i(R_j, t) \times Pdemand(A_i) \quad \textbf{(EQ 11)}$$

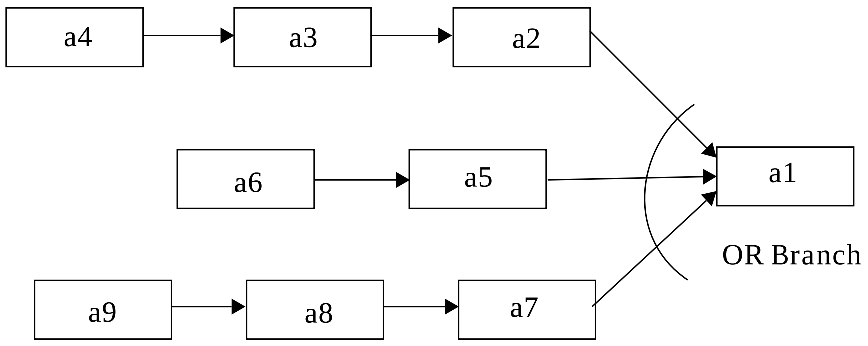
Thus, the final overall individual demand looks like:

$$\overset{\circ}{D_i}(R_j, t) = D_i(R_j, t) \times \frac{1}{alt(A_i)} \times amt(A_i) \times Pdemand(A_i) \quad \textbf{(EQ 12)}$$

4.4.2.1 Texture Measure on OR Constraints/

Revisiting our texture measures, there are two fundamental textures. *Reliance* of activities on resources is the first one. An aggregation of this gives us *contention* on resources, which the second texture being used. This section presents another texture measure, at a yet higher aggregation, defined on OR constraints. The context for this is the framework of a disjunctive process plan, described in chapter three in detail. Each such process plan is a network of activities, and one of the decisions to be made is to assign a percentage-demand to each activity. In effect, this is equivalent to assigning a percentage demand to each linear branch at all branching nodes in the process plan. For an illustration, refer to the figure below.

Figure 13. Branching Node in a Process-plan



It is evident from the figure that assigning a percentage to activity *a3* has the effect of assigning the same percentage to *a2* and *a4*, since they belong in the same *linear* path. In fact, this is a flow / conservation constraint, explained in Chapter 3 and in Section 4.4.4.1. In other words, at each OR constraint, we are not assigning percentages to activities, but to entire paths. Thus, this decision cannot be made on the basis of information on a single activity alone. We have to consider all the branches / paths involved in the decision.

In order to do that, we introduce the concept of **criticality**: *activity criticality* and *path criticality*. The criticality of an activity reflects its overall importance in the schedule. It combines the demand effects of all resources with which an activity interacts. Consider activity *a2* to have more than one resource request variable (*rrv*). Further, consider each *rrv* to have a non-unit resource domain. Then, the criticality of an activity, $Cr(A_i)$ is given by:

$$Cr(A_i) = \sum_i D_{rrv_i}^{aggr} \qquad \textbf{(EQ 13)}$$

where $D_{rrv}{}^{aggr}$ is the aggregated demand for a resource request variable, given by:

$$D_{rrv_i}^{aggr} = \sum_j D_{R_j}^{aggr}(t) \qquad \textbf{(EQ 14)}$$

where $D^{aggr}{}_R$ is the aggregated demand (contention) on a resource, given by (EQ 5). The criticality of an activity will give a true picture of the importance of that activity over the entire schedule. However, even this computation does not give us the entire picture for each *path*. To obtain that, we need to extend this reasoning further.

We propose that the criticality of a linear path is equal to the criticality of the *most* critical activity on that path. An alternative reasoning here could be to use the sum of all activity criticalities on a path to obtain path criticality. However, we believe that stronger results will be observed by relieving the tightest clique of constraints, i.e., most critical activity. Thus when faced with an OR constraint (percentage-demand) decision, we need to traverse each linear branch and isolate the most critical activity on each path, thus giving us path criticalities. This is a new texture measure proposed in this work. Its use is made in variable selection in our Policy 1, explained shortly.

4.4.3 Inventory Reasoning

ODO version1 had no explicit representation of or reasoning about inventory. We have extended this version to address that shortcoming. The representational aspects have been already presented in chapter three. In this section, we will discuss implementational or problem solving details. In any given process plan, there may be a number of activities that consume or produce inventory. In fact, activities are often *linked* to each other through inventory; the product of one activity is the raw-material of another. Apart from this, there are two more issues of concern for inventory requests within our model.

- amount / quantity / number of units required
- quality of inventory; specifically, code-age requirements.

Thus each inventory request requires a specific product, specific quantity, and specific quality. Linking each inventory request to the correct product is an inherent part of our process plan design. We have divided the remaining functionality among two policies. The first ‘prunes’ out the inventory (object-sets) that do not meet the age criteria. The second allocates remaining inventory to the requests according to quantity requirements.

4.4.3.1 Inventory Availability Profile

Referring back to (EQ 9), aggregate demand curve is normalized using resource capacities. For non-inventory resources, capacity is considered constant over the duration of the schedule. Evidently, the same does not hold for inventory, which is periodically consumed and produced. Therefore, we have to first compute inventory capacity, or *availability* curve. This is complicated by the fact that activities are not scheduled in time yet. We only have a window for activity; thus, the measures used are probabilistic rather than deterministic. This implies that the approach to be used is similar in concept to that of computing demand profiles.

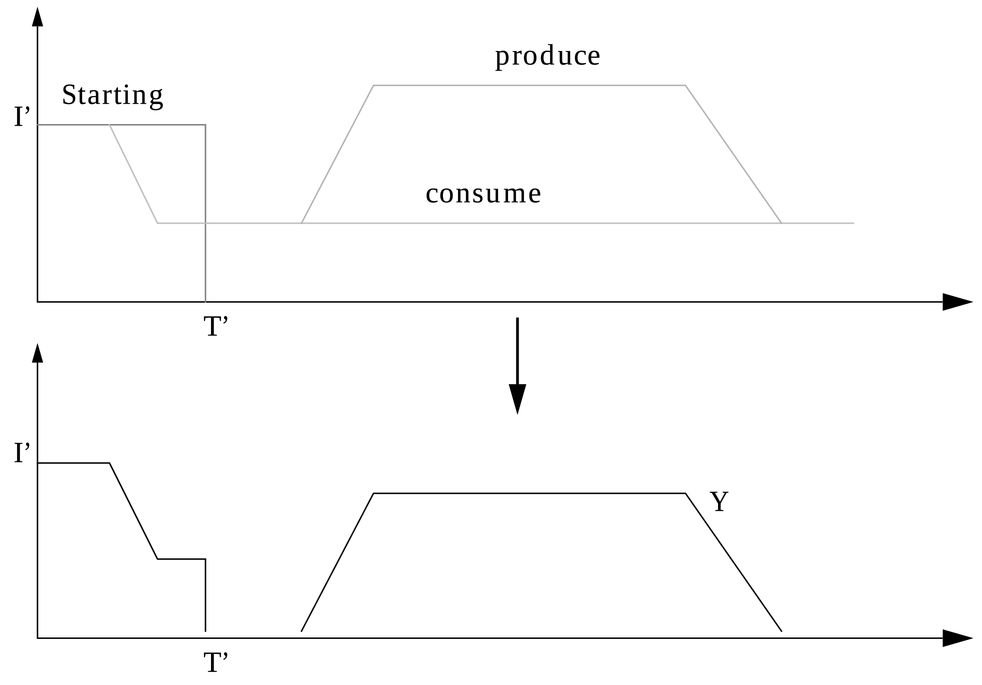
We look at the inventory in an aggregate fashion, and obtain the consumption / production data for products. This is used to build the probabilistic availability profile for each product. What we use to construct this curve on products is: base individual-demand curve (reliance) on activities, amount of a product required by activity, shelf-life of products, and time windows on activities. Consider the following:

- Activity *A1*, which consumes *X* of product *P*, with required shelf life *t*. Consider *A2* which produces *Y* of *P*. *P* has shelf life of time *T* units.
- Also consider starting inventory *I'* for product P. The shelf-life remaining of this is *T'*.

We build the individual demand curves on activities as discussed earlier. These curves are utilized in the following manner in order to compute the availability curve of product *P*:

1. Multiply each probabilistic demand value by the amount required by an activity. This gives us the probabilistic *amount* required.
2. For the activity which produces *P*, extend the curve rightward by time *T* (shelf-life of produced inventory). This means that this inventory is available (if no other activity consumes it) until at most *latest-end* + *T*, after which it spoils.
3. For the consume activity, invert the curve to reflect consumption. Also, extend the curve rightward, to reflect the fact that this amount is not returned after the activity has executed.
4. For starting inventory, start the curve with the Y-axis value at *I'*, which goes to zero at time 0+*T'*.
5. Add (algebraically) the modified curves for *A1* and *A2* to the starting inventory curve, in order to get the availability curve of the product *P*.

Figure 14. Availability Curve on Product P

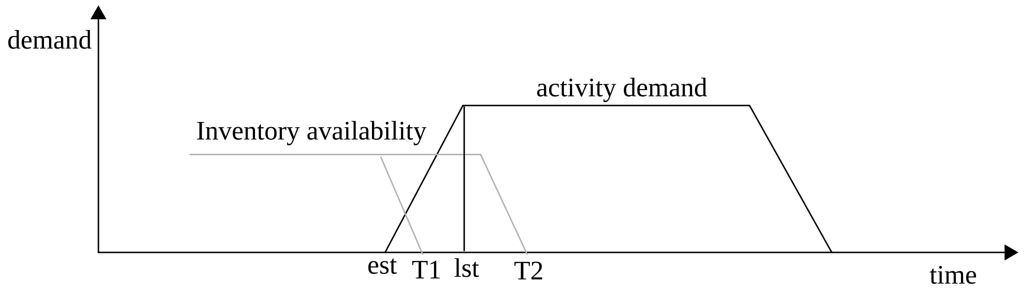


The aggregate demand curves on inventory are computed as for other resources. Once we have both the availability curve and the demand curve, we can obtain the normalized demand curve. Each point of the aggregate demand curve is divided by the corresponding point of the availability curve for this normalization.

4.4.3.2 Related Constraints

This section discusses two sets of constraints which are specific to managing inventory. Each of these is encapsulated in a separate policy. The first of these is the **code-age** constraint, which constrains the quality of products for different orders. Code-age period is the remaining time/life of a product before it spoils. A code-age constraint is a property of an order, and specifies the minimum code-age of allocated products. Policy 2 performs code-age reasoning for all activities which require inventory towards an order. All inventory which does not meet the criteria is removed from the domain. The constraint is implemented as illustrated below, using activity time window and product shelf-life.

Figure 15. Code-age Constraint



Looking at the inventory availability profile, we see time points T1 and T2. T2 represents the time-point of that product spoiling. T1 is a function of an order, and represents T2 minus code-age required. Thus, with respect to that order, the inventory effectively spoils at time T1. It should be noted that at this point, activities do not have a fixed starting time. Thus, in order for inventory to be acceptable, T1 should always be at least as high on the timeline as lst. In that case, whatever time the activity starts, the inventory is feasible. Thus in the above illustration, the inventory represented is not acceptable.

The second set of constraints drives Policy 3, by relating the total **quantity** of a product in the 'available' domain of an activity to the quantity of that product required by that activity. This is more involved than non-inventory resources. Unlike resource requirement which is satisfied with exactly one resource, an inventory request is often satisfied by a combination. The constraint simply states that the total amount of all inventory (over all object sets in the domain) an activity has access to is greater than or equal to the amount required by the activity. The constraint can be presented as:

$$Q(a_i) \le \sum_{1}^{n-1} Q(R_j) \qquad \textbf{(EQ 15)}$$

where n-1 is the remaining number of object-sets, after removing the contended one.

4.4.4 Disjunctive Process-plans: Further Reasoning

We have already explained much reasoning behind our treatment of network-like process plans. Specifically, the heuristic reasoning for variable selection has been explained. This section discusses the next step: value selection and assertion. Variable selection in this case gives us an activity with unassigned **status** value. Thus value selection involves assigning a value (percentage) to that variable.

However, since this process is guided by *three* sets of constraints and the design of the *status variable*, we discuss these first. The `status` of an activity is integral, and can assume values from that activity's `min-status` to its `max-status`. The following relations exist among these values:

$$0 \leq minstatus \leq status \leq maxstatus \leq 100 \qquad \textbf{(EQ 16)}$$

Two sets of constraints are **flow/conservation** constraints, which conserve the flow of material through a network of activities. The first is defined over all linear paths in a process-plan, and is called the *equality constraint*. This constraint states that "*the status values for all activities on the same linear path in a process plan are equal.*" With reference to Figure 13, "Branching Node in a Process-plan," on page 77, this constraint can be represented as:

$$status(a_2) = status(a_3) = status(a_4) \qquad \textbf{(EQ 17)}$$

The second set is defined over the branching node, and defines the relationship between the status of a parent path and that of its children branches. This is the *sum-equals* constraint.[1] The constraint states that "*the sum of the status values of all the branching activities is equal to the status value to the branching-node predecessor activity.*" Again, with reference to the same figure as above, the mathematical form would be:

$$status(a_2) + status(a_5) + status(a_7) = status(a_1) \qquad \textbf{(EQ 18)}$$

The use of the above variables and constraints conserves the flow of materials through the supply chain, and also keeps the network consistent with respect to demand. In other words, these concepts are used both in value selection as well as value assertion.

The core of **value selection** lies in the sum-equals constraint, (EQ 18). At this point, we already have a variable, which is one of the activities on one of the three branches. For the sake of clarity, let us assume it is one of *a2*, *a5* and *a7*. We have to assign a value to the status variable of that activity, a process which will also change the domains of the status variables of the other two activities. Let us further assume that *a2* is the most critical activity (variable selected) and denote *status*(a_i) by s_i. (EQ 18) can be rewritten and transformed as:

1. The form of this constraint is similar to Kirchoff's Law of electricity.

$$s_2 + s_5 + s_7 = 1 \quad \text{(EQ 19)}$$

which a simple linear equality. Referring back to the structure of the status variable set, we can introduce additional constraints through the bounds/domains (`min-status`, `max-status`) of the status variables. Thus for each s_i, we get the following constraint:

$$min(s_i) \le s_i \le max(s_i) \quad \text{(EQ 20)}$$

The objective of this heuristic decision is to reduce the overall contention in the network. Activities with high criticalities represent tight constraint cliques. Thus our aim is to minimize the overall criticality level at a branching node. This is gives us an objective function to minimize:

$$min\left(\sum_{1}^{n} Cr_i(s_i)\right) \quad \text{(EQ 21)}$$

where Cr_i is the criticality of activity a_i. Thus, value selection is not reduced to a *linear program* given by (EQ 19) through (EQ 21). Solving this LP gives us the status value to be assigned to the selected activity. The complete procedure is presented in Appendix X.

Value **assertion** involves changes to the selected activity and this is executed using the third set of constraints, *production-rate* constraint. This constraint is also explained in Section 3.2.3.1 on page 38. It relates the status value of an activity to its variable duration. Thus as the status value changes, it is reflected in the changed duration of the corresponding activity. An increase in status (or, percentage-demand) results in a corresponding increase in the duration of that activity, through this constraint, in order to satisfy increased demand. This concept is similar to that of continuous activities.

$$Prate(a_i) \times duration(a_i) = D(a_i) \times D(a_j) \quad \text{(EQ 22)}$$

$D(a_i)$ is the percentage (demand) of the selected activity (say a_2) while $D(a_j)$ is the total demand required through its parent branching node (a_1 in this case).

Value **propagation** is performed after assertion, and the purpose of this process is to make the network consistent after the assignment of status value to one variable. There are two distinct procedures of interest here.

- re-balancing the disjunctive/conjunctive branching node (sum-equals constraint)
- consistency along a linear path (equality constraint)

Both of these are performed *recursively* from the decision point in the process plan to its “leaf” activities. Once an activity is assigned, the ‘balance’ of status is distributed as before among the remaining activities. Referring to the above example (EQ 19), say we assign `status` = 0.2 to activity a_2. The remaining activities’ status is:

$$s_5 + s_7 = s_1 - s_2 = s_1 - 0.2 \qquad \text{(EQ 23)}$$

This is distributed between a_5 and a_7 as before, typically equally or equiprobably. Thus after each value assertion, the corresponding node is balanced.

After this procedure, we re-enforce consistency along each of the three linear paths; this is governed by the equality constraint, (EQ 17). This means that:

$$s_4 = s_3 = s_2 = 0.2 \qquad \text{(EQ 24)}$$

and likewise for all other linear paths. Recursiveness comes into play when any of these linear path branches out when moving downward (opposite the flow of material). Then we again perform both of these procedures, and so on.

At this point, we have provided adequate explanations behind most of our complex reasoning. The next section presents our actual problem solving policies in which the above reasoning is embedded.

4.5 Problem Solving Policies

In order to have an efficient algorithm which minimizes backtracking and finds good solutions, it is important to design superior variable/value ‘ordering’ heuristics. [Sadeh 91] has reported excellent results with variable/value ordering heuristics based on contention/reliance measures. In this section we present our variable/value ordering heuristics, which are built using our contention/reliance based texture measures, presented in the previous section. The overall goal of our heuristics is to reduce contention on resources, i.e., reduce the aggregate demand on resources. Each type and instance of commitment we make works towards reducing contention on one or more resources. Since our problem is a satisfaction problem, it is our assertion that reducing contention across the board will lead us to one or more satisfying solutions. Reducing demand is also equivalent to repairing capacity violations in the solution. Thus lowering resource contention in the problem moves us towards a satisfactory solution.

In terms of the *overall solution process,* five policies are executed sequentially. The algorithm works in the following manner:

1. The complete problem and the solving policies (heuristics) are input to the scheduler.

2. Pre-processing occurs before any heuristic can be executed. This includes establishing initial arc consistency, resource consistency, and the initial activity network.

3. The first policy to be executed assigns a percentage-demand to each activity. Thus, we start with a "floating" activity network, and this heuristic "fixes" this network. After this step, we have a fixed / firm network, with weights assigned to each activity which specify what percentage of the demand (through a process plan) the activity satisfies.

4. The second policy performs inventory level reasoning on the basis of code-age requirements. Activities requiring inventory are selected at random and for each, all inventory (object-sets) which do not satisfy that activity's code-age requirements are pruned from its domain.

5. The third policy also performs inventory level reasoning, from the perspective of quantity. For all activities, one or more object-sets are allocated to each such that each activity has the required amount, and each object-set is feasible with respect to capacity.

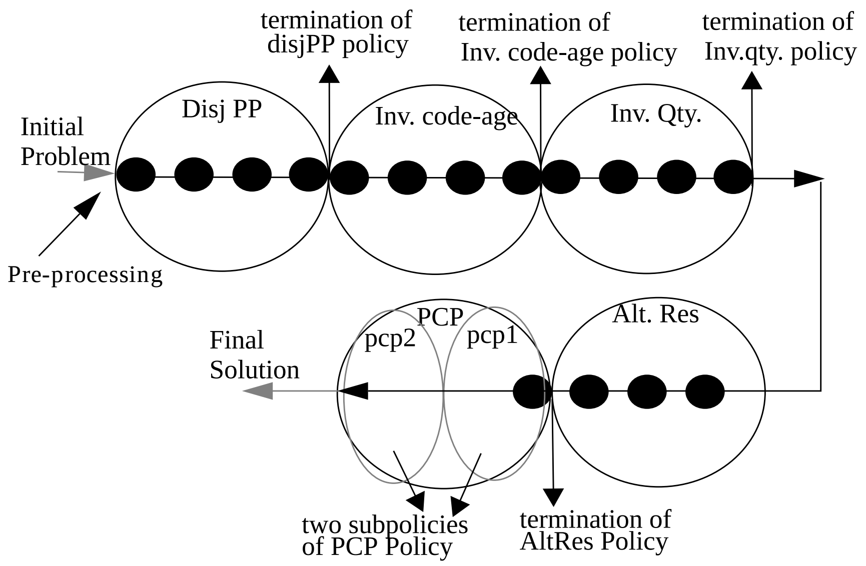
6. The next policy works on another level of disjunctiveness, that of each activity to be executed having a set of alternative (non-inventory) resources to pick *one* from. This set of heuristics prunes the set of resources for all activities down to one value only. In other words, at the end of this policy, all activities have one specific resource assigned to them.

7. The last policy sequences all the activities (with best possible resource assignments) in time. As mentioned in Section 1.1 on page 1, our algorithm does not assign fixed start times to activities. Instead, we partially sequence the activities such that there are no

capacity violations. Thus, at the end of this policy, if the problem is solved, we have a feasible window for all activities. In other words, we arrive at a *family* of solutions. If the problem is not solved, the policy gives us the least-cost solution it can find.

Further, each of these policies may themselves be composed of more than one sub-policy. This cycle is illustrated in the following figure:

Figure 16. Flow of Control


termination of disjPP policy
termination of Inv. code-age policy
termination of Inv.qty. policy
Disj PP
Inv. code-age
Inv. Qty.
Initial Problem
Pre-processing
PCP
pcp2
pcp1
Alt. Res
Final Solution
two subpolicies of PCP Policy
termination of AltRes Policy

Prior to presenting policies, we introduce ODO's policy declaration mechanism. This would enable us to present a complete PODL declaration of each policy as we explain it.

4.5.1 ODO Policy Declaration

The Policy is the actual embodiment of the heuristic-based algorithm that is operating on the constraint-based representation of a scheduling problem. As such it is based on the selection and retraction of commitments. Policies also provide the main control functionality in the problem solving process.

A real embodiment of a particular scheduling heuristic is defined as an **atomic-policy**, in contrast to **meta-policy**, which is the control structure of the Policy hierarchy. It is very much possible for us to employ more than one heuristic techniques on one problem. Each of these techniques is encapsulated in one atomic-policy, which has its own termination criteria. A meta-policy keeps executing its children (in order) until it’s own termination criteria are met.

4.5.1.1 MetaPolicy

The MetaPolicy is the control structure of the Policy hierarchy. It keeps executing its children (in order) until it’s own termination criteria are met In our algorithm, a meta-policy terminates once its has executed all its atomic-policies exactly once. To declare a MetaPolicy, a meta-policy `<command>` is used. A sub-BNF for the meta-policy `<command>>` is as follows:

```
(meta-policy :name <word>
      :state-acceptance-criteria <word>
      :state-cost-function <word>
      :termination-criteria <word>
      :sub-policies (<word> <word> ..)
)
```

- :name <word>

  Associate a unique identifier with the MetaPolicy.

- :state-acceptance-criteria <word>

  A condition to determine whether the evaluated state is accepted (thus going forward) or rejected (thus backtrack). Current acceptable criteria are:

  - `always`: always accept the new state.
  - `cost-leq`: only accept the state if its *cost* is less than or equal to that of the previous state.
  - `no-empty-pvs`: only accept the state if the (start-time) possible-value set for all activities is non-empty.

- :state-cost-function <word>

  A function to evaluate the resulting state after propagation. Current acceptable functions are:

  - `num-realDemand-contended-resources-early`: the number of resources which are contended for in the current state, on the basis of real demand and earliest start time assignment of requesting activities.
  - `num-non-unit-rrvs`: number of resource requests which have non-unit domains.

- :termination-criteria <boolean-expression>

  A condition to determine whether to terminate search using this policy.

  - cost, <integer>, search-time, num-backtracks, iterations, search-exhausted, <, >, >=, <=, ==,!=, ||, &&

- :sub-policies <word> <word> ..

  Specifies the child atomic-policies associated with this meta-policy. Each <word> is the name of a unique atomic-policy.

4.5.1.2 Atomic-Policy

The Atomic-Policy is the embodiment of a particular scheduling heuristic. As such it can make only one type of commitment. It has a number of doubly-linked lists of functions used to make, retract, and propagate commitments. To declare an Atomic-Policy, an atomic-policy `<command>` is used. A complete sub-BNF for the atomic-policy `<command>`> follows. Some components which are not explained in detail are not part of our algorithm, but are part of the generic policy declaration mechanism of ODO. Also, examples are not provided here but are given with the actual policy descriptions.

```
(atomic-policy :name <word>
      :commitment-type <word>
      :forward-commitments
           (filters :generate <list>
                    :select <list>
                    :score <word>
                    :select-scored <list>)
      :backward-commitments
           (filters :generate <list>
                    :select <list>
                    :score <word>
                    :select-scored <list>)
      :propagation-methods <list>
      :backtrack-method <word>
      :state-acceptance-criteria <word>
      :state-cost-function <word>
      :termination-criteria <word>
)
```

- :name <word>

  Associates a unique identifier with the AtomicPolicy.

- :commitment-type <word>

  Specifies the type of commitment that is being executed.

- :forward-commitments
  - `:generate <list>`

    Based on the list of filters, generate one commitment to be made at this step.

- `:select <list>`

  Defines functions to select a value to commit for the commitment instance which was generated at the above `generate` step.

- :backward-commitments
  - `:generate <list>`

    Defines the functions to generate/select a commitment to retract at the backtrack step.

- :propagation-methods <list>

  Defines the propagation method(s) which guide the propagation after each commitment is made.

- :backtrack-method <word>

  A release procedure to follow if the resulting state is rejected by the evaluation.

- :termination-criteria <boolean-expression>

  A condition to determine whether to terminate search using this policy.

  - cost, <integer>, search-time, num-backtracks, iterations, search-exhausted, <, >, >=, <=, ==, !=, ||, &&

4.5.2 Pre-Processing Details

In this section, we present some *pre*-processing details, i.e., the processing of a problem before actual scheduling starts. Of course, full *arc consistency* is performed to ensure that the problem is temporally feasible. Similarly, *resource completeness* check is performed to see if the problem is resource consistent. That is, an activity may not request a resource which has not been declared. Such a case results in pre-processing error and problem solving termination.

Processing of *orders* and *process plans* is also fairly complex. Each order instantiates one or more “copies” of the corresponding process plan, based on its required amount. The system computes this and creates the activity network corresponding to each order. Further, as mentioned earlier, the *durations* of these activities are not fixed; each activity has a window, the bounds of which correspond to the minimum and the maximum demand it can satisfy. Based on the pre-defined and undefined demand values for each activity, the pre-processing routines set the corresponding durations or windows. The temporal bounds on each activity sub-network are also computed from the due-date of the order it represents.

4.5.3 Policy 1: Disjunctive/Conjunctive Process Plans

We start the scheduling process with a set of disjunctive/conjunctive process plans, i.e., our original activity network has a number of *and*, *or*, and *and/or* relations between activities. This branching arises from the fact that in many process plans, there is more than one sequence to obtain the final product. Each sequence usually is composed of different activities. Now, it becomes a scheduling decision to select one or more sequences in a given process plan. Further, in case more than one path is selected, another scheduling decision is to allocate a percentage of the demand to each of those paths, such that the end result is the order quantity of the process plan.

Thus, the scheduling decision at each step then becomes assigning a demand to an activity, as a percentage of the total demand being met through its parent process plan. A related feature of this approach is that the user can control of this process. Thus, a user can specify these percentages for some or all activities, in which case the algorithm will not alter them. For all activities where the user does not assign percentages, the algorithm does so, with the goal of reducing overall contention on the resources in the system.

The next question is, what actually happens when these percentages are altered? Value assertion and propagation has been explained in detail in Section 4.4.4. With each such decision, the activity network is altered and has to be made consistent again. Demand consistency is ensured by using *equality* and *sum-equals* constraints. However, the activities are also altered with change of percentage demands.

The major question here is how to reflect the change of demand for a given, single activity? How can one activity represent production from zero to 100% of demand? We model this change using *variable durations* and a *production rate* on activities. The representational concepts are explained in Section 3.2.3 on page 38, whereas the nature of the guiding constraint is presented in Section 4.4.4. This constraint links the percentage demand, production rate, and the duration of each activity. Thus, when the percentage-demand is increased, the duration of the corresponding activity also increases proportionately. Therefore, the “longer” activity can now handle the increased demand placed on it. This model quite closely reflects the notion of continuous activities also.

Thus, if we decrease the demand on one activity (in order to reduce contention on the resource it requires), we invariably have to increase the demand on a parallel activity, and its duration. This is a trade-off which has to be considered, and one which requires more research into. The heuristics of this commitment type assign a percentage demand to each activity which is not fixed by the user, such that overall contention on resources is reduced as much as possible.

4.5.3.1 Variable Ordering

The goal of this variable ordering heuristic is to select the most *critical* activity at this iteration. The following steps are executed:

- Compute the individual demand curves of activities, and then aggregate demand on resources.
- For each activity whose status is not fixed yet, compute criticality using (EQ 13), (EQ 14), and the aggregate demand on resources.
- Select the activity with highest criticality; break ties arbitrarily.

4.5.3.2 Value Ordering

This heuristic accepts the most critical activity and assigns a value to its status variable after solving a pseudo-LP. The steps involved are:

- Consider the local network where the selected activity is situated. Specifically, consider the linear path where the selected activity is, the parallel linear paths, and their junction activity. Refer to Figure 13, “Branching Node in a Process-plan,” on page 77.
- Traverse each linear path (except the one of the selected activity) in this local network and find the most critical activity on each.
- Using these activities, construct a LP as explained in Section 4.4.4 on page 81. Solve the LP to obtain the value for the status variable of the selected activity.

- Assign this value to the variable, and using the production-rate constraint, alter the duration of the activity to reflect this.

4.5.3.3 Assertion/Propagation

The network is made consistent with respect to demand recursively using the equality and the sum-equals sets of constraints. This is explained in detail in Section 4.4.4 on page 81. This also alters the duration of the activity involved in each iteration. This means that internal temporal propagation has to be performed in order to make the internal (start, duration, end) variables consistent again. There is no external propagation since the duration window is not affected.

4.5.3.4 Termination Criteria

The goal of this heuristic is to transform a “probable” network into a fixed network, i.e., to assign a fixed percentage demand to all activities. Thus this policy terminates when all initially unassigned activities have been assigned a percentage demand between zero and 100.

4.5.3.5 Policy Specification

```
(atomic-policy :name fix-process-plan
      :commitment-type assign-percentage-demand
      :forward-commitments
          (filters :generate most-critical-OR
                   :select fix-least-critical-branch)
      :propagation-methods percentage-demand-propagation
      :state-acceptance-criteria always
      :state-cost-function any-status-change
      :termination-criteria cost==0
)
```

4.5.4 Policy 2: Code-Age Reasoning

Code-age period is the remaining time/life of a product before it spoils. A code-age constraint is a property of an order, and specifies the minimum code-age of allocated products. This policy performs code-age reasoning for all activities which require inventory towards an order. All inventory which does not meet the criteria is removed from the domain. The constraint is implemented as illustrated Section 4.4.3 on page 78.

4.5.4.1 Variable Ordering

This heuristic is quite simple and selects a previously not selected activity randomly. Since all activities are to be made code-age compatible independently, ordering is trivial.

4.5.4.2 Value Ordering

The goal of this heuristic is to prune the domain (corresponding to an inventory request) by removing all resource / object sets which do not satisfy the code-age constraint. The reasoning has been explained in Section 4.4.3 on page 78. The steps are:

- Consider the individual demand curve of the selected activity.
- For each set in the domain, compute the code-age expiry time. If this time is less than the latest-end time of the activity, remove this set from the domain.
- Stop when all sets in the domain have been reviewed.

4.5.4.3 Termination

There is no propagation in this policy. We terminate when all activities (and all inventory requests) have been considered.

4.5.4.4 Policy Specification

```
(atomic-policy :name code-age-reasoning
      :commitment-type remove-spoilable-sets
      :forward-commitments
           (filters :generate unfiltered-activity
                        :select remove-spoilable-sets)
      :propagation-methods none
      :state-acceptance-criteria always
      :state-cost-function number-of-activities-with-spoilable-sets
      :termination-criteria cost==0
)
```

4.5.5 Policy 3: Inventory Allocation

After code-age requirements have been met, specific inventory (or, specific object sets) has still to be allocated to activities’ inventory requests. This task is more involved than when performed for non-inventory resources, as in Policy 4. The reason is that unlike assigning a single machine to an activity out of a pool of say five, this policy typically will have to satisfy each inventory request through a combination of multiple sets.

4.5.5.1 Variable Ordering

The goal of this heuristic is to select the activity which most heavily relies on the most-contended for inventory (object-set). The steps are:

- Compute the individual demand profiles for all activities; then compute aggregate demand curves for all object-sets.
- Select the most-contended-for set
- Select all activities which demand this set, and sort them in the order of reliance, with the most reliant activity being the first.

4.5.5.2 Value Ordering

This heuristic takes a least commitment approach and removes the most-contended-for set from the domain of the most reliant activity only, subject to (EQ 15). The steps are:

- Accept the sorted list of activities and pick the topmost one.
- Remove the most-contended-for set from that activity's domain, subject to the constraint that the remaining quantity in the domain is greater than or equal to that required.
- If the constraint is violated, leave the first activity and pick the next one. Check for constraint (EQ 15) again.
- Repeat these steps until one activity's domain is pruned.

4.5.5.3 Termination

Again, no propagation is required. This policy terminates when for all activities, the quantity of inventory is domain is (approximately) equal to that required. Termination may also occur earlier, if all object-sets are feasible with respect to capacity; i.e., no sets is violated in terms of capacity.

4.5.5.4 Policy Specification

```
(atomic-policy :name allocate-inventory
      :commitment-type filter-resource-set
      :forward-commitments
           (filters :generate most-contended-for-set
                        :select most-reliant-activity)
      :propagation-methods none
      :state-acceptance-criteria always
      :state-cost-function any-set-contended-for
      :termination-criteria cost==0
)
```

4.5.6 Policy 4: Alternate Resources

Consider an activity network where each activity requires one or more resources, and has a set of resources to choose one from, for each requirement. In other words, each resource request of an activity has alternatives to select from. The fact that an activity may conjunctively require more than one resource does not have a bearing on this commitment. Thus, the variables here are all resource requests which have a non-unit domain. Assigning a value to them means assigning a single resource to them.

Rather than assign one resource to a request, we perform a finer search and follow a least commitment approach. We do this by not assigning a resource to a request at each iteration, but by pruning an alternative from the domain of a request, at each iteration.

4.5.6.1 Variable Ordering

The goal of our variable ordering heuristic is to select the activity which contributes most to the current bottleneck resource. The following steps are executed:

- Compute the aggregate demand curves of all resources any request for which still has alternatives.
- Compute the most-contended-for interval on each curve, and get the sum of that demand.
- Select the interval which has the maximum demand on it. This is the current point of contention.
- Generate a list of all activities contributing to that contention formation.
- Sort these activities according to their individual demand contributions to this contention, from highest to lowest contribution.
- Return the topmost activity in the list which has a non-unit domain.

We have implemented two versions of this heuristic: one based on probabilistic demand, and the other on real (actual) demand.

4.5.6.2 Value Ordering

Value ordering in this case means the pruning of the resource domain of the request returned by the variable ordering heuristic. This decision is straight-forward here; the current most-contended-for resource is removed from the domain of the selected request. The logic here is that by un-assigning the “critical” activity from the current contention, we relieve maximum contention on the bottleneck.

4.5.6.3 Termination Criteria

Since the goal of this algorithm is to assign one specific resource to each request, that defines the termination criteria. The search terminates once all request by all activities have exactly one resource assigned to them.

4.5.6.4 Policy Specification

```
(atomic-policy :name assign-resources
      :commitment-type filter-resource
      :forward-commitments
          (filters :generate most-contended-for-resource
                      :select most-reliant-activity)
      :propagation-methods none
      :state-acceptance-criteria always
      :state-cost-function number-of-non-unit-domain-activities
      :termination-criteria cost==0
)
```

4.5.7 Policy 5: Precedence Constraint Posting

Consider all as of yet unordered activity pairs, a pair being two activities requesting the same resource, without any temporal constraint between them. Each such pair can also be viewed as an ordering *O(i,j)*, with no value assigned to it. A value assignment in this case would be a sequence, or a precedence constraint between the two activities. Thus, our variables in this commitment type are these possible orderings in the problem. In each iteration, we select one unassigned variable (pair) and assign a value (precedence) to it. It is to be noted here that the aim is not to order all possible pairs. Rather, we order only as many pairs as required to remove any capacity violations from all resources.

4.5.7.1 Variable Ordering

The goal of our variable ordering heuristic in this case is to select two activities A_i and A_j, which contribute most heavily to the current contention, and return them as an unassigned ordering *O(i,j)*. The following steps are executed:

- For all contended-for resources, compute their aggregate demand curves.
- Compute the most-contended-for interval on each curve, and get the sum of that demand.
- Select the interval which has the maximum demand on it. This is the most-contended-for resource and interval.
- Generate a list of all activities contributing to that contention formation.
- Sort these activities according to their individual demand contributions to this contention, from highest to lowest contribution.
- Select the two topmost activities, i.e., two activities which contribute the most to the current contention. A temporal constraint will be posted between these activities.

  Note: Our algorithm prevents any cycles in the graph by performing a complete check on these activities for any existing linkage through any partial network. If these two activities are already linked via another path, select another activity, until we find an unlinked pair or exhaust all pairs. In case we exhaust all pairs, and the resource is still contended-for, the problem is infeasible.

Again, we have implemented two versions of this heuristic: one that performs all computations using probabilistic demand, and the other that uses real/actual demand.

4.5.7.2 Value Ordering

A variable *O(i,j)* can take one of two values: O_1*(i-->j)* or O_2*(j-->i)*. A value ordering heuristic should select the value which is most likely to survive future assignments. In this case, since we are sequencing activities, the above could be interpreted to mean that we select a value which is most compatible with the currently existing or “tending-towards” ordering. Thus our assertion is that the value which preserves any explicit or implicit orderings in the current search state is most likely to guide us to a solution.

One component of our value assignment is [Erschler 76] work on Constraint Based Analysis (CBA). CBA attempts to identify some natural orderings in the search space, based on the temporal variables of activities forming the pairs. Following is a brief synopsis of this technique. Let us consider two activities A_i and A_j, with early start times est_i and est_j, latest end times lft_i and lft_j, and durations du_i and du_j respectively. Consider the following three relationships between these variables:

$$\lambda = lft_i - eft_j \quad \text{(EQ 25)}$$

$$\mu = lft_j - est_i \quad \text{(EQ 26)}$$

$$\delta = du_i + du_j \quad \text{(EQ 27)}$$

Then, for any unsequenced pair (*i,j*), we can distinguish four cases:

1. If $\lambda < \delta <= \mu$, then *i-->j*

2. If $\mu < \delta <= \lambda$, *j-->i*

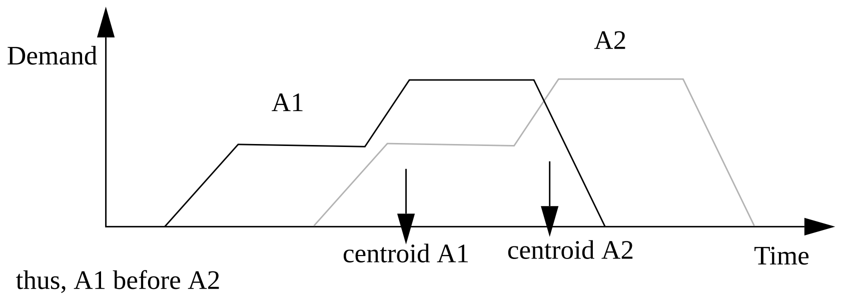
3. If $\delta > \mu$ and $\delta > \lambda$, then no feasible solution is possible

4. If $\delta <= \mu$ and $\delta <= \lambda$, then either sequencing decision is still possible

Case 4 becomes interesting since this is where the heuristic search is defined. We have implemented three heuristics for this case, where each of them preserves the implicit ordering, although based on a different criteria. These are:

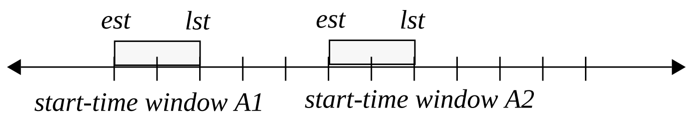
1. **Demand Centroid**: each activity in the pair has an individual demand curve on that resource. We compute the centroids of the two curves, and sequence the activities according to the position of their demand centroids.

Figure 17. Demand Centroid based Value Selection



2. **Earliest Start**: this is a simpler approach, which looks at the earliest-start-time (*est*) of the two activities, and preserves that ordering.

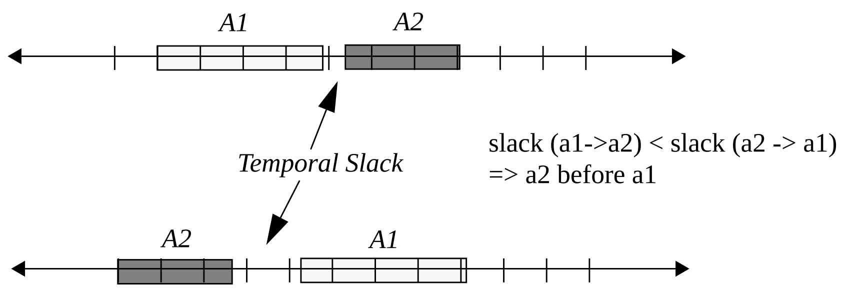
Figure 18. Early-start based Value Selection



Thus, A1 before A2

3. **Temporal Slack**: here, we perform a lookahead to see which ordering (*i-->j* or *j-->i*) leaves a larger temporal slack on the resource. We select the ordering which leaves more slack, by the reason that more slack means a lesser constrained resource.

Figure 19. Temporal Slack based Value Selection



4.5.7.3 Termination Criteria

The goal of this algorithm is to sequence activities on each resource in order to remove all capacity violations. Thus, our termination criteria uses real demand on resources and real capacities, and measures the difference. The algorithm terminates when real demand is less than or equal to capacity, for all resources.

4.5.7.4 Propagation

This policy sequences activities in time, and thus requires complete temporal propagation. After each iteration, we perform forward and backward propagation from the two activities respectively. This guarantees network consistency after propagation, if the network was arc consistent before the commitment. The routines also perform internal propagation for all activities.

4.5.7.5 Policy Specification

```
(atomic-policy :name precedence-constraint-posting
      :commitment-type post-precedence-constraints
      :forward-commitments
           (filters :generate [high-demand-pair, high-real-demand-pair]
                      :select [CBADemandCentroid,CBAEarlyStart,
                                CBATemporalSlack])
      :backward-commitments
           (filters :generate most-recent-failure)
      :propagation-methods chronological
      :state-acceptance-criteria always
      :state-cost-function number-of-contended-resources
      :termination-criteria cost==0
)
```

4.6 Conclusion

In this chapter, we have presented in detail our problem solving algorithm in terms of its component heuristics, its execution loop, commitment types, termination criteria, and so on. We have also discussed what pre-processing is performed in ODO. Also presented is the implementation level of our algorithm, in terms of PODL input, heuristic filters and the policy mechanism.

The next chapter discusses our experiments using ODO as a supply chain scheduler.