NOTICE

The following document is the most complete specification of problem definition in the PODL language that exists. It is (as will be seen) part of a to-be-completed thesis and therefore we have some trepidation about releasing it to the world.

Therefore, you are granted access to this document under the following conditions:

- it is used solely for education / research purposes.
- it is not reproduced in any form unless this notice is attached. Please keep reproductions to a minimum, though we don't expect you to not photocopy it.
- you may not reference it in any papers.

When the thesis is complete (soon) you will be able to do all the things you can normally do with a thesis. If you would like to be notified when the thesis is published, send email to chris@ie.utoronto.ca.

Please direct comments and questions to Chris Beck (chris@ie.utoronto.ca) and leave Sanket alone.

Chapter 3 Representing Problems in ODO

This chapter describes our representation of the supply chain scheduling problem and how we model it in ODO. We also present **PODL** (**O**DO **P**roblem **D**escription **L**anguage) and show how to specify our problem in PODL, for input to ODO. This chapter begins with a discussion of the problem model of ODO: Version I, which provided the initial basis for this work.

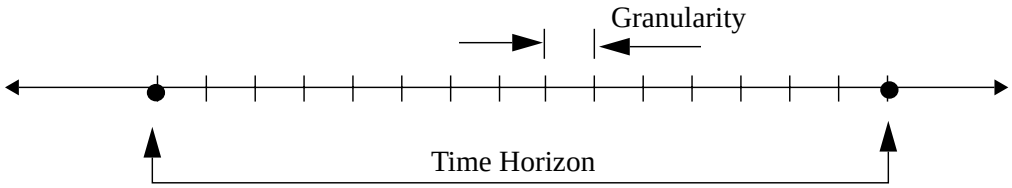
3.1 ODO Version I: Problem Representation

This section presents the problem representation 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. Thus its problem model combines the essential elements of a job-shop scheduling problem into a generic representation which is presented below.

3.1.1 Time

Time is an integral component of the scheduling problem. ODO represents three concepts about time: continuity, intervals, and, instances. Continuity is represented by a *time-line*, which is theoretically infinite in both directions. This line is composed of numerous instances or *time-points.* An *interval* defines the region that lies between two non-identical time-points. Refer to the following figure for an illustration.

Figure 4. Timeline

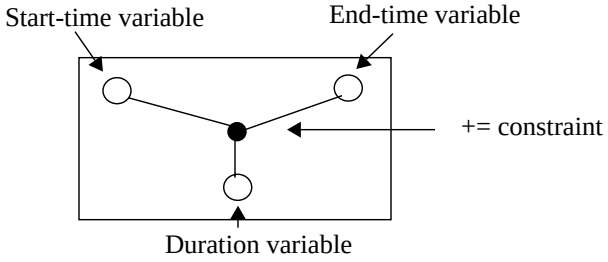


A special type of interval is called *time-horizon*, which is a characteristic of scheduling / planning problems. A horizon defines the interval within which all events occur; no event can occur outside the scheduling / planning horizon. A number of other components relate to time, and are appropriately named. For instance, *temporal variables* (start / end times of activities), *temporal constraints* (precedence), and so on. In the current implementation of ODO, the granularity is one time unit and the horizon is defined by the minimum / maximum times declared for activities.

3.1.2 Jobs/Activities/Tasks

ODO represents a job-shop scheduling problem with *jobs* being composed of *activities*. Collectively, these may simply be referred to as tasks. All tasks have a fixed *duration*, which is the difference between its *start* and *end* time point. Thus, a task is defined by three temporal variables: start-time, end-time, and duration.

Figure 5. Job/Activity



3.1.3 Resources

Resources are entities like facilities, raw materials and personnel which are used or consumed by activities during their execution. An actual resource which is used is called a *resource-pool*. Resource pools are aggregated by *resource-classes*, which define the functionality

of a pool. Each resource-pool is attached to the timeline via its *history*, which represents the availability of the pool with time.

ODO models two types of resources: those that are used (machines) and those that are consumed (raw materials). Both of these are declared identically using resource classes and pools, parameterized by an integral capacity / availability. The system restricts a user to declare the parent resource class before a member pool is declared.

3.1.4 Temporal Constraints

Temporal constraints connect temporal variables, for example, the duration constraint connects start and end time variables (Figure 5, "Job / Activity," on page 33). Others include constraints which connect different activities (before / after) and over schedule constraints like scheduling horizon. Constraint may also restrict the domains of temporal variables, for example *earliest-start / latest-end* times of activities.To represent temporal relationships between activities, ODO model all 13 of Allen's [Allen 84] temporal relations. For example, the relation a2 starts after a1 ends is modeled as:

$$st(a2) >= et(a1) \quad \textbf{(EQ 1)}$$

3.1.5 Resource Constraints

Resource constraints connect the resources required by an activity to that activity. Each requirement may be parameterized by the resource class required and the amount required. Also, each activity may require more than one resource. Each such requirement is encapsulated within a *resource request variable* attached to an activity. A resource constraint ties one resource request variable (which will be assigned a resource pool as its value) to the start and end times of the activity. The constraint is satisfied only if the assigned resource pool has sufficient capacity to support the activity. This information is extracted from resource history. Further, for each use / consume / produce of a resource, the corresponding history is updated to reflect that allocation.

Job-shop scheduling problems are constructed within ODO using the above concepts. In place of constraints and variables, ODO uses an input language which is naturally descriptive of the problem, similar in concept to ODO Version II's input language, presented later in this chapter. However, a description of version I's input language is outside the scope of this work.

3.2 Problem Representation and Declaration: Version II

Since our problem also belongs to a class of scheduling problems, it has many familiar components like time, activities, resources, and constraints. Activities use / consume / produce resources, and are bound by start and end time intervals. The duration of each activity is also an interval. Resources can play three different *roles* in the problem: mechanism, material, or, container. There are many types of constraints, like: temporal, capacity, flow, shelf-life, and so on. We also model more involved concepts like network-like process-plans, orders, and shelf-life / code-age of products. *Code-age* is customer specific, and specifies the minimum *remaining* shelf-life required of a product. In the following part of this chapter, use of the name ODO refers to **our** version and not version I.

ODO has a textual interface for specifying **scheduling** problems and corresponding solving strategies. This is called **PODL** (**O**DO **P**roblem **D**escription **L**anguage), pronounced *poodle*. It is very user-*un*friendly to describe a problem in terms of (lower level) variables and constraints, or by equalities / inequalities. PODL represents a very intuitive way of specifying scheduling problems, in terms of resources / activities / constraints and so on.

Furthermore, PODL is designed to be flexible, extensible and incremental. Not only can we specify many classes of scheduling problems, we can also aim to specify, say, MRP type problems, with some extensions. This section presents a step-by-step tutorial on how to declare a supply chain scheduling problem in PODL.

The language (PODL) has a LISPish feel: parentheses denote a command in the language. This method has been adopted to provide the user with a very simple method of grouping the parameters of an activity. This format is also very flexible so that a user can omit certain parameters completely and define them in any order. The general syntax is:

(<command> :arg_name1 arg_value1 :arg_name2 arg_value2 …)

Many of the arguments have a default value, so that if it is not specified, the default is used. The order in which the parameters appear is not important.

The BNF syntax of PODL follows. We assume the usual definitions of `<ascii>`, `<alphabetic>`, `<numeric>`, `<backslash>`, and `<whitespace>`. The '*' character indicates 0 or more occurrences, braces, '{}' indicate the scope of the grammatical operator.

<PODL_LANGUAGE> ::= {<command> <whitespace>}*
<command> ::= (<word> {<whitespace> :<word> <whitespace> <expression>}*)
<expression> ::= <atom> | <list> | <command>
<atom> ::= <word> | <string> | <number>
<word> ::= <alphabetic><character>*
<character> ::= <alphabetic> | <numeric> | <special>
<number> ::= <numeric>* | <numeric>*.<numberic>*
<string> ::= "" | "<initial-component>*<final-component>"
<initial-component> ::= <ascii>-<double-quote> | \"
<final-component> ::= <ascii>-<double-quote>-<backslash> | \"
<list> ::= (<whitespace>* {<expression> <whitespace>}*)

The following sections are grouped by object. Each section describes the information we model, its corresponding PODL syntax, and presents an example declaration.

3.2.1 Time

Time is an integral component of the problem. We represent three concepts about time: continuity, intervals, and, instances. Continuity is represented by a *time-line*, which is theoretically infinite in both directions. This line is composed of numerous instances, which we call *time-points*. An *interval* defines the region that lies between two non-identical time-points.

A special type of interval is called *time-horizon*, which is a characteristic of scheduling / planning problems. A horizon defines the interval within which all events occur; no event can occur outside the scheduling / planning horizon. A number of other components relate to time, and are appropriately named. For instance, *temporal variables* (start / end times of activities), *temporal constraints* (precedence constraints), and so on.

3.2.2 Schedule

In a sense, a schedule is the parent object of all other objects in a problem. All activities, resources, constraints, etc. *belong* to a particular schedule. Each schedule has a corresponding time-horizon, which defines the start and the end time of that schedule. All events occur between these two time-points.

A schedule object has to be created before any activity / resource is defined. To declare a schedule, a schedule `<command>` is used. A sub-BNF for the schedule `<command>>` is as follows:

```
(schedule :name <word>
      :min-time <atom>
      :max-time <atom>)
```

- :name <word>

  Associates a unique identifier with the schedule.

- :min-time <atom>

  Specifies the lower bound of the time window that the activities are to be scheduled on.

- :max-time <atom>

  Specifies the upper bound of the time window that the activities are to be scheduled on.

An example schedule object declaration is:

```
(schedule :name myexample
      :min-time 0
      :max-time 100)
```

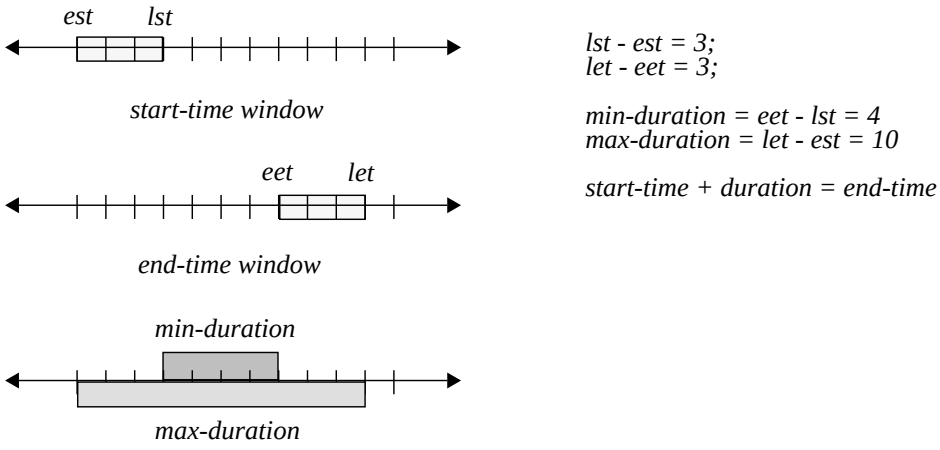
3.2.3 Activities

3.2.3.1 Activity Description

The problem comprises of assigning resources to activities within time requirements. Thus activities are entities which should be assigned feasible resources and time-slots, for a feasible problem solution. Activities are often aggregate, i.e., composed of many other *sub*-activities.

All activities have three temporal attributes: *start-time*, *duration*, *end-time*, each of which can be represented by a time-interval. Therefore, we can have earliest-start-time and latest-start-time, earliest-end-time and latest-end-time, and, minimum-duration and maximum-duration. On a time-line, start-time plus duration equals end-time. A final solution typically replaces all windows by fixed points (st, et) and interval (duration). These concepts are illustrated in Figure 6, “Activity Representation,” on page 38.

Figure 6. Activity Representation



It may also be possible for a final solution to leave all temporal variables with feasible intervals, in which case we have a family of solutions. Activities use / consume / produce resources; these relationships are implemented through resource / capacity constraints. An activity can request a specific resource, a specific sub-set of resources, or, all resources which

play a certain role. For example, consider 10 milling machines named M1 to M10. An activity may request M3, or it may ask for (M1,M2,M3), or, it could simply specify ‘milling-machine’, in which case, all 10 machines are acceptable.

Further, in the supply chain scheduling problem, each activity belongs to a specific process-plan instance (refer to Section 3.2.4 on page 46). A process plan being a conjunctive / disjunctive network of activities, each activity may satisfy all, or only a part of the total demand being satisfied by the parent process plan. This is represented as the *demand* attribute of each activity, expressed as a percentage of the total demand being satisfied by the parent process plan.

As another extension, we first state that all activities consume / produce resources. In the case of *continuous* activities, consumption / production is performed at a certain *rate*. We are in a position to enhance our representation by introducing this concept within. Given the fact that we model percentage demands and variable durations, we can link the two using a *production-rate*, as shown below. Since the demand represents amount produced, we use only a production rate.

$$Prate(a_i) \times duration(a_i) = demand(a_i) \quad \textbf{(EQ 2)}$$

where *demand(a_i)* gives the total demand satisfied by activity a_i.

3.2.3.2 Activity Declaration

The activity **command** defines a schedulable object. The definition includes a description of the resource requirements of the activity and its temporal relations to other activities. To declare an activity an activity `<command>` is used. A sub-BNF for the activity `<command>>` is as follows:

```
(activity :name <word>
      :earliest-start <atom>
      :latest-end <atom>
      :min-start <atom>
      :max-start <atom>
      :min-end <atom>
      :max-end <atom>
      :duration <atom>
      :min-duration <atom>
```

```
      :max-duration <atom>
      :demand <atom>
      :min-demand <atom>
      :max-demand <atom>
      :rate <atom>
      :temporal-relation (<t-relation> :activity <word>
                                       :interval <atom>
                                       :time-point <atom>)
      :uses (object :object-name <word>
                    :role-name <word>
                    :amount <atom>)
)

<t-relation> ::= after | before | equals | meets | met-by |
                 overlaps | overlapping | during | contains |
                 starts | started-by | finishes | finished-by
```

- :name <word>

  Associates a unique identifier with the activity.

- `:earliest-start <atom>`

  The earliest time at which the activity may start. This is typically used in an aggregate activity to specify the release time for an order.

- :latest-end <atom>

  The latest time which an activity may end. Used with `:earliest-start`.

- `:min-start <atom>`

  The activity starts no earlier than this time point.

- `:max-start <atom>`

  The activity starts no later than this time point.

- `:min-end <atom>`

  The activity ends no earlier than this time point.

- `:max-end <atom>`

  The activity ends no later than this time point.

- `:duration <atom>`

  The length of time that an activity takes to complete.

- `:min-duration <atom>`

  The minimum length of time that an activity takes to complete.

- `:max-duration <atom>`

The maximum length of time that an activity takes to complete.

- :demand <atom>

  The demand an activity satisfies, as a percentage of the demand satisfied by its parent process plan. The process plan being a network, each activity may have alternatives, which satisfy the remaining demand. See Section 3.2.4 on page 46.

- :min-demand <atom>

  The minimum demand an activity satisfies, as a percentage of the demand satisfied by its parent process plan.

- :max-demand <atom>

  The maximum demand an activity satisfies, as a percentage of the demand satisfied by its parent process plan.

- :production-rate <atom>

  The production rate of an activity, defined as amount produced per unit of time.

- :temporal-relation <command>

  See Section 3.2.3.2.1 on page 41.

- :uses <command>

  See Section 3.2.3.2.2 on page 44.

3.2.3.2.1 Temporal Relation Grammar

PODL supports each of Allen's [Allen 84] temporal relationships. To define a temporal relation between activity B and an already defined activity A, the parameter `:temporal-relation` is used. The parameter value for the temporal relation is a `<command>` itself of the form:

```
(activity :name B       :min-duration <atom>       :max-duration <atom>
      :temporal-relation (<t-relation> :activity A
                                       :interval <atom>
                                       :time-point <atom>))
```

- <t-relation>

Defines the type of temporal relation that the activity that is begin declared (activity B in the above example) has with the already defined activity identified in the `:activity` parameter (in the above example, A).[1] The following are valid `<t-relations>` (we will use activities A and B declared as above to provide examples of each type of t-relation):

- `after`

  The start-time of Activity B must be greater-than the end-time of Activity A. Namely, `B after A: AAA BBB`

- `before`

  The end-time of Activity B must be less-than the start-time of Activity A. Namely, `B before A: BBB AAA`

- `equals`

  Both the start-time and end-time of Activity B must be respectively equal to the start-time and end-time of Activity A.

  Namely, `B equals A: AAA`

  BBB

- `meets`

  The end-time of Activity B must equal to the start-time of Activity A. Namely, `B meets A: BBBAAA`

- `met-by`

  The start-time of Activity B must be equal to the end-time of Activity A.

  Namely, `B met-by A: AAABBB`

- `overlaps`

  The start-time of Activity B must be less-than the start-time of Activity A and the end-time of Activity B must be less-than the end-time of Activity A. (In other words, Activity A begins after B begins and ends after B ends).

  Namely, `B overlaps A:` `BBB`
  AAA

- `overlapped-by`

  The start-time of Activity B must be greater-than the start-time of Activity A and the end-time of Activity B must be greater-than the end-time of Activity A.

  Namely, `B overlapped-by A:` `BBB`
  AAA

- `during`

1. The requirement that the other activity is already declared does not pose any limitations on the temporal relations that can be defined. As described, every temporal relation also has an inverse (e.g. meets and met-by).

The start-time of Activity B must be greater-than the start-time of Activity A and the end-time of Activity B must be less-than the end-time of Activity A.

Namely, `B during A:`

```
  BBB
AAAAAA
```

- `contains`

  The start-time of Activity B must be less-than the start-time of Activity A and the end-time of Activity B must be greater-than the end-time of Activity A.

  Namely, `B contains A:`

```
BBBBBB
  AAA
```

- `starts`

  The start-time of Activity B must be equal to the start-time of Activity A and the end-time of Activity B must be less-than the end-time of Activity A.

  Namely, `B starts A:`

```
BBB
AAAAA
```

- `started-by`

  The start-time of Activity B must be equal to the start-time of Activity A and the end-time of Activity B must be greater-than the end-time of Activity A.

  Namely, `B started-by A:`

```
AAAAA
BBB
```

- `finishes`

  The end-time of Activity B must be equal to the end-time of Activity A and the start-time of Activity B must be greater-then the start-time of Activity A.

  Namely, `B finishes A:`

```
  BBB
AAAAA
```

- `finished-by`

  The end-time of Activity B must be equal to the end-time of Activity A and the start-time of Activity B must be less-then the start-time of Activity A.

  Namely, `B finished-by A:`

```
AAAAA
  BBB
```

There are three possible parameters inside the `:temporal-relation <command>`. These are as follows:

- :activity

  Defines the activity with which the temporal relationship is shared.

- :interval

Modulates the meaning of the <t-relation> by defining an interval of time. For example, if the after <t-relation> is used and the :interval is defined to be 10, the relation has the meaning that the end-time of activity B must be greater-than the end-time of Activity A plus 10 time points. So if A ends at 25, B cannot begin until at least 36. We do *not* implement variable intervals, i.e., intervals defined by min/max values.

Note that :interval doesn't apply to all of Allen's relations. It makes sense only to the following relations: after, before, overlaps and overlapped-by. For after and before relations, the :interval is the minimum time elapse between the two activities in concern. As for overlaps and overlapped-by, :interval is the overlapping time interval when both activities are being executed.

- :time-point

  PODL can also express a temporal relation between an activity and a time point. For example, if the <t-relation> is contains and the :time-point is 30, it specifies that activity B is constrained to be executing at time point 30. Specifying a time-point is semantically well-formed for a sub-set of the Allen relations: after, before, meets, met-by, contains, started-by, finished-by.

3.2.3.2.2 Resource Requirements Grammar

PODL supports resource usage/consumption/production, and specification of required capacity.

```
(activity :name B:min-duration <atom>:max-duration <atom>
   :uses (object :object-name <word>
      :role-name <word>
      :amount <atom>)
   :uses (role :role-name <word>
      :amount <atom>)
   :consumes (resource-set :set-name <word>
      :role-name <word>
      :amount <atom>)
   :consumes (role :role-name <word>
      :amount <atom>)
   :produces (resource-set :set-name <word>
      :role-name <word>
      :amount <atom>)
)
```

There are four parameters within the commands shown above. These are:

- :object-name

  Defines the name of the object which this activity uses.

- set-name

  Defines the name of the resource-set which this activity consumes / produces.

- :role-name

  Defines the role which the required object / set is supposed to play in this activity.

- :amount

  Gives the amount of object / set required by this activity.

3.2.3.3 Activity Examples

Two example activity declarations are:

```
(activity :name activity1
      :min-start 10
      :max-end 25
      :duration 10
      :demand 40
      :production-rate 5
      :temporal-relation (before :activity activity2
                                          :interval 5)
      :uses (object :object-name MM1
                    :role-name millingmachine
                    :amount 1)
      :consumes (role :role-name rawstock
                    :amount 10)
      :produces (resource-set :set-name stock1
                    :role-name steelstock
                    :amount 20)
)

(activity :name activity2
      :min-start 10
      :max-start 25
      :min-end 50
      :max-end 65
      :duration 40
      :min-demand 50
      :max-demand 100
      :production-rate 25
      :temporal-relation (before :activity activity5
                                          :interval 0)
      :uses (object :object-name LM2
                    :role-name lathemachine
                    :amount 2)
```

```
        :consumes (role :role-name steelstock
                        :amount 20)
        :consumes (role :role-name coolant
                        :amount 3)
        :produces (resource-set :set-name widget10
                        :role-name widget
                        :amount 100)
)
```

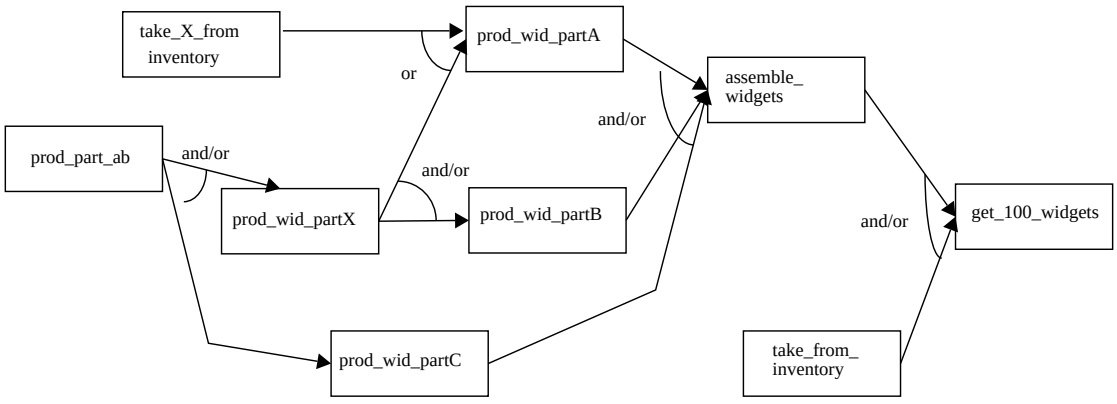
3.2.4 Process Plans

3.2.4.1 Process Plan Description

A process-plan is defined for a product, and represents a *network* of activities which when performed will result in availability of that product. In other words, a process-plan can also be called a way to satisfy demand for the product. We represent process-plans as "activity templates", or, as **generic** pre-defined networks of activities. These templates are *instantiated* for products required, thus resulting in activity *network(s)*. Since actual problem solving is performed on activities, we do not model attributes other that activity-sequence in process-plans.

Process plans are tree like routings, with branches that are both conjunctive and disjunctive. Simple cases would be linear and fixed tree-like process plans. However, a realistic process plan of a supply chain is a dynamic network of activities, the final structure of which is developed through the solution process. The following figure shows such a process plan:

Figure 7. Supply-Chain Process Plan



The **OR** decision points represent conjunctive / disjunctive decisions, whereas the **XOR** represents a pure disjunct. For example, to satisfy an order for 100 widgets, we may assemble 100 widgets, or take 100 widgets from inventory, or, assemble some and take the rest from inventory. Thus, the final structure of this network will be determined by the solution process, based on some criteria. This process is described in detail in chapter 4. However, in brief, activities at an OR / XOR branch are assigned percentage demands such that the sum of these is 100. The goal of this process is to find an assignment which results in the least possible contention on the resources required by participating activities. After each such assignment, these percentages are propagated across the network to ensure consistency with all upstream activities.

3.2.4.2 Process Plan Declaration

A process-plan for a product represents a generic sequence of activities to be performed to obtain (manufacture / purchase) an Order Quantity (OQ) of that product. This order quantity is fixed, and pre-defined. The conjunctive / disjunctive branching, as discussed in Section 3.1.2.1 on page 31, is built into the syntax of an activity. **All** activities that comprise a process plan are declared within this <command>. Following is the sub-BNF for the process-plan <command>:

```
(process-plan  :name <word>
               :product-name <word>
               :order-quantity <atom>
               :has-activity <activity <command>>
               :has-activity <activity <command>>
               :.......
               :has-activity <activity <command>>
)
```

We have already discussed <activity <command>> in Section 3.2.2 on page 37. Activities declared as part of a process plan do not have start / end times associated with them. The bounds on these are provided by the order release / end times, and through temporal propagation. The other attributes are:

- :name

Defines a unique name for this process-plan.

- :product-name

  Defines the product (set-with-role) which this process-plan produces. This name corresponds to a resource-role played by a resource-set, in our model.

- :order-quantity

  An integer, defining the number of units of the product manufactured by *one* instantiation of the process-plan.

3.2.4.3 Process Plan Example

A simple example process plan which transforms raw stock into widgets is:

```
(process-plan  :name makewidget
      :product-name widget
      :order-quantity 100
      :has-activity (activity
         :name activity1 :min-start 10 :max-end 25 :duration 10 :demand 100
         :temporal-relation (before :activity activity2 :interval 5)
         :uses (object :object-name MM1
                       :role-name millingmachine
                       :amount 1)
         :consumes (role :role-name rawstock
                       :amount 10)
         :produces (resource-set :set-name stock1
                       :role-name steelstock
                       :amount 20)
)
   :has-activity (activity
         :name activity2 :min-start 10 :max-start 25 :min-end 50
         :max-end  65 :duration 40 :min-demand 100 :max-demand 100
         :uses (object :object-name LM2
                       :role-name lathemachine
                       :amount 2)
         :consumes (role :role-name steelstock
                       :amount 20)
         :consumes (role :role-name coolant
                       :amount 3)
         :produces (resource-set :set-name widget10
                       :role-name widget
                       :amount 100)

)
```

3.2.5 Resources

3.2.5.1 Resource Related Description

Resources are entities which are used / consumed / produced / required by activities within the problem. Some examples of resources are, machines, facilities, people, inventory, trucks, and so on. We have represented resources in a very generic, flexible, and extensible manner. We model resources as entities or **objects**, which can then play certain **roles** in activities.

As an example, consider a milling-machine. It is modeled as an object called machine1, which plays the role of milling-machine in activities. In another case, consider a conveyer-belt in a warehouse. It is an object which is continuously playing two roles, one of a transportation device, and another of a storage facility.

To represent inventory, we use an *aggregation* of objects, which we call a **object-set**. Thus, a box of 100 pencils (objects) is a object-set. Sets also play certain roles in activities, and are similarly required by activities.

Objects / sets also have a representation of *capacities*, which are modeled for a specific role. The variation of capacity with time is represented by the *history* of the resource again for a specific role. Both of these issues are discussed in next two sections.

3.2.5.2 Resource Roles

We recognize three *classes* of resource roles in our model. These are:

- *MechanismRole*: all roles which are a mechanism in the system, i.e., these roles are played by resources *used* by activities, e.g., machines, warehouses, plants, trucks, people.
- *MaterialRole*: all roles which are a material in the system, i.e., roles played by resources *consumed / produced* by activities, e.g., inventory, products.
- *ContainerRole*: the resources playing this role are able to *contain* other resources within them. For instance, a warehouse (resource) contains inventory (resource).

In our implementation, we have imparted MechanismRole and ContainerRole to objects, and MaterialRole to object-sets. The design however is flexible enough to allow for any extensions. We also model what we call a *resource-role-class*, which represents an aggregation of

role instances. For example, all instances of MechanismRole which are called MillingMachine belong to a resource-role-class called MillingMachine. Thus we can access all these instances through a parent role class.

3.2.5.3 Capacity

Both objects and resource-sets have some representation of capacity / availability, but through the roles they play. This is because of the obvious fact that the capacity displayed by a resource can vary with role. For instance, the capacity of a truck in its role as transportation would probably be its throughput (X units of resource carried / hour), whereas it would be in cubic feet, for container-role. These different notions of capacity could be inter-related, which is not a subject of research here.

We have modeled three “types” of capacity / availability:

- *Simultaneity Capacity*: modeled for MechanismRole, this represents the number of activities which a resource can support simultaneously.
- *Set-size*: modeled for resource sets (MaterialRole), it can also be called discrete availability. This represents the number of units (objects) available within a set.
- *Storage Capacity*: modeled for ContainerRole, represents the amount of resources which can be contained within the resource playing this role. A common unit (like cub. m) is used to model storage capacity and requirements.

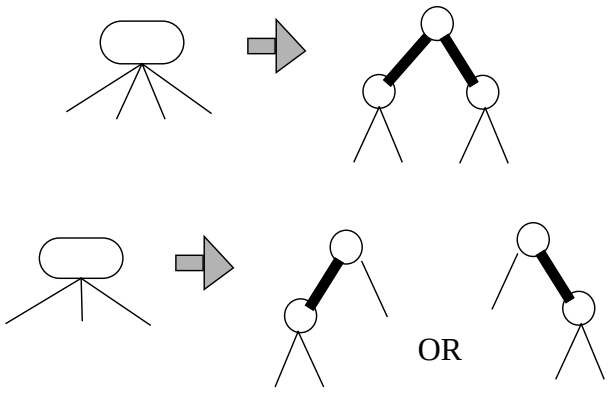
The mapping of role-type and capacity is as shown below:

Object/set	**role-type**	**amount (capacity type)**
object	Mechanism	simultaneity capacity
object	Container	storage capacity
object-set	Material	set-size (discrete availability)

The variation of capacity with time is called the **history** of a resource, with a specific role. We have implemented histories using *red-black-trees*, a variation of balanced binary trees (refer to Figure 8, “Red-Black-Trees,” on page 51). Each history instance is one tree, and each history element (also a tree element) stores a time interval, and the capacity value during that interval.

Red-Black-Trees are formed when 2-3-4 trees (balanced binary trees) are represented as standard binary trees, in the following way. The idea is to represent the 3-nodes and 4-nodes are small binary trees bound together by “red” links, as opposed to the regular “black” links of the 2-3-4 tree.

Figure 8. Red-Black-Trees



3.2.5.4 Resource Declaration

3.2.5.4.1 Resource Role Class

Each resource plays a role in an activity, which is declared within the resource requirement grammar of the activity. Prior to declaring any role instance, we have to declare its parent resource-role-class.

```
(role-class :name <word>
   :role-type <word>
   :shelf-life <atom>
)
```

The parameters of the class declaration are:

- :name

  Gives the name of the resource-role-class, and of subsequent child (same name) resource-role instances.

- :role-type

  This refers to classes of roles, as presented in Section 3.2.5.2 on page 49. Each resource-role-class instance is of one of these role types.

- :shelf-life

  Defines the shelf-life for all child role instances with `role-type` MaterialRole.

Two examples of declaring a role-class are:

```
(role-class :name millingmachine :role-type Mechanism
)
(role-class :name cheese :role-type material :shelf-life 28
)
```

3.2.5.4.2 Object With Role

An object `<command>` instantiates a new object within the schedule, and sets role(s) for that object to play.

```
(object :name <word>
   :has-role (role :name <word>
          :sim-capacity <atom>)
   :has-role (role :name <word>
          :storage-capacity <atom>)
)
```

- :name

  Gives the unique name of that object.

- `:has-role`
  - `:name`

    Defines the name of the role object is to play.

  - `:sim-capacity`

    Defines the simultaneity capacity for that role instance (if it is type Mechanism), as an integer value.

  - `:storage-capacity`

    Defines the storage capacity for that role instance (if it is type Container), as an integer value.

Two example declarations of an object are:

```
(object :name Machine1
   :has-role (role :name millingmachine
          :sim-capacity 6)
(object :name storage-area-1
```

```
   :has-role (role :name storagearea
          :storage-capacity 600)
```

3.2.5.4.3 Resource Sets

A resource-set <command> instantiates a new set in the schedule, quite like the object <command>.

```
(resource-set:name <word>
             :age <atom>
   :has-role (role :name <word>
          :amount <atom>)
)
```

- :name

  Gives the unique name of that resource-set.

- :age

  Gives the current age of that resource-set, in number of time units.

- :has-role

  - :name

    Defines the name of the role that set is to play.

  - **:**amount

    Defines the *initial* amount available for that resource-set. This can also be considered the starting inventory.

An example declaration of a resource-set is:

```
(resource-set :name cheese-10
   :age 20
   :has-role (role :name cheese
          :amount 125)
```

3.2.6 Demand (Orders)

Demand for products within the systems is represented by **orders** for these products. An order is for a specific product (resource with MaterialRole), has a *due-date* and a *code-age* associated with it. Each order can potentially be satisfied through one or more process-plans.

ODO instantiates one or more process-plans for each order, and builds up the activity network for the problem.

The order `<command>` declares a new order within the schedule. One schedule may have more than one order.

```
(order :name <word>
       :product-name <word>
       :quantity <atom>
       :due-date <atom>
       :code-age <atom>
       :latest-end <atom>
       :early-release <atom>
)
```

The sub-BNF is:

- :name

  Defines a unique name for this order.

- :product-name

  Defines the product (set-with-role) for which this order is placed.

- :quantity

  An integer, defining the number of units of the product required.

- :due-date

  An integer, it defines the number of time units (from current) within which the order should ideally be satisfied. This gives us our due-date constraint.

- :code-age

  An integer which defines the code-age value required of allocated sets (product). This gives us the code-age constraint on the order.

- :latest-end

  An integer, it defines the number of time units (from current) within which the order must be satisfied. This gives us our end-time constraint. Due-date is less that or equal to end-time. The difference in these two is a measure of order *tardiness*. It is equal to due-date, by default.

- :early-release

An integer, it defines the number of time units (from current) until which the order cannot commence (to be satisfied). This gives us our earliest-start constraint. The default value is equal to current time.

A *sample* order is given below:

```
(order :name order-101
       :product-name widget
       :quantity 200
       :due-date 30
       :code-age 20
       :latest-end 32
       :early-release 0
)
```

3.2.7 Constraints

3.2.7.1 Temporal Constraints

Temporal constraints are those related to time, and connect temporal variables in the constraint graph. The constraint on the *duration* of an activity is a temporal constraint, as are *precedence* constraints and restrictions of *start/end* times of activities.

We represent all of Allen's [Allen 84] temporal relations between activities, individually (refer to Section 3.2.3.2.1 on page 41). For example, the relation "strictly after" is implemented as a *less-than* constraint between end-time of one activity and start-time of another. The duration and start/end time constraints restrict the set of possible values for these variables.

Temporal constraints are posted though declarations of orders, activities, and so on. For example:

```
(order :name order-101
       :latest-end 32
       :early-release 0
       :.....
)
```

The above order declaration places constraints of the start/end times of process plans (and thus activities) which satisfy this order. Also,

```
(activity :name activity2
      :.....
      :temporal-relation (before :activity activity5
                                          :interval 0)
      :.....
)
```

The above activity declaration results in posting of a precedence constraint between two activities.

3.2.7.2 Resource And Capacity Constraints

All activities require certain resource for their execution. Constraints which specify the resources required, their quantities, requirement duration, and so on are collectively called resource constraints. For instance, an activity called 'delivery' will require a truck, a driver, some fuel, some time, and maybe other resources.

Constraints specifying/limiting the consumption/use of resources are called capacity constraints. Capacity could be in terms of number of units available, rates of consumption, and others. For example, the maximum capacity of a warehouse could be $100m^3$, minimum being zero.

Capacity/availability constraints govern the present capacity and utilization of a resource. The capacity available is declared within the declaration of an object/set with-role. This value defines the maximum capacity for that object/set *in* that role, and is also represented in the *history* of a resource.

These constraints are posted via resource/activity declarations. For example, declaration of resources constrains their maximal capacities.

```
(resource-set :name cheese-10
   :age 20
   :has-role (role :name cheese
          :amount 125)
(object :name storage-area-1
   :has-role (role :name storagearea
```

```
        :storage-capacity 600)
```

On the other hand, activity declarations post constraints not only linking activities and resources, but also modifying resource capacities.

```
(activity :name activity2
      :min-start 10
      :max-start 25
      :min-end 50
      :max-end 65
      :duration 40
      :min-demand 50
      :max-demand 100
      :temporal-relation (before :activity activity5
                                              :interval 0)
      :uses (object :object-name LM2
                    :role-name lathemachine
                    :amount 2)
      :consumes (role :role-name steelstock
                    :amount 20)
      :consumes (role :role-name coolant
                    :amount 3)
      :produces (resource-set :set-name widget10
                    :role-name widget
                    :amount 100)
)
```

The above declaration links `activity2` to `LM2`, `steelstock`, `coolant`, and `widget`. Further, it specifies that 2 units of `LM2` may be used from time 10 to 65. Also, 20 units of `steelstock` and 3 units of `coolant` will be consumed at time 10, whereas 100 `widgets` will be produced at time 65.

3.2.7.3 Flow/Conservation Constraints

Supply chain is concerned with the flow of material from net producers to net consumers. These constraints ensure that the material flowing through an enterprise is conserved, i.e., consumption and production balance each other in a stable system. For example, all the raw materials consumed in the system are accounted for in the orders satisfied by the system, plus any leftover inventory. These constraints actually exist in any physical supply chain system; the question is how to model them?

These constraints are built into our definition of process plans, and the definition of demand related constraints within disjunctive process plans. For example, the sum of the product flowing through different branches is proportionately equal to the amount of material being consumed at one end, or produced at the other. Also, the amount of product flowing through a linear section of a process plan is conserved. The simple example in Section 3.2.4.3 on page 48 shows how a process plan is designed to ensure conservation.

3.2.7.4 Product Constraints

Certain constraints affect only products, and are quite independent of the associated activities. These are grouped under the heading product constraints. The exact composition of a product in terms of other products (a Bill of Materials) is a product constraint. In essence, these constraints tie the various products and consumables together in some fashion. Again, the definition of process plans, and how different inventories contribute towards a final product, depend on these constraints.

3.2.7.5 Order Constraints

This category includes specific constraints imposed by customers, through their orders. For example, the “due-date” of an order is such a constraint. Another example is a “code-age” constraint, which defines the minimum age of the products(s) assigned to an order.

3.2.7.6 Priority Constraints

Priorities attached to different components of a system are represented as soft or hard constraints. For instance, high priority may be attached to consumption of inventory, rather than fresh production. This may have the effect of forcing the system to consume inventory, and avoid situations like stock-piles, spoilage, and so on.

There are a number of ways to attach priorities in our model, although we generally view priorities as ways to implement rules-of-thumb, which we do not wish to make a part of our solver. However, a user can specify certain priorities through the use of percentage demands on various activities, to manipulate the final structure of process plans, and the usage of various resources. Consider the following two activities connected by an *OR* constraint:

```
(activity :name a1
      :min-demand 20
      :max-demand 100
      :......
)

(activity :name a2
      :min-demand 0
      :max-demand 80
      :......
)
```

Quite clearly, we have attached a preference on `a1`, by giving it a higher `min-demand` than `a2`. Whatever the situation may be, we will always execute `a1`, but may not execute `a2`.

3.2.7.7 Shelf-Life/Code-Age Constraints

These can be regarded as *composite* constraints, i.e., they have *both* temporal and resource aspects. We have defined the above mentioned concepts in Section 1.3.4 on page 15. Each order declares its code-age requirement, which gives us a bound on the age of products to be allocated to this order, thus posting a constraint.

Each resource-set plays a role, which has a shelf-life associated with it. For example, a role called ‘widget’ may have a shelf-life of 12 weeks; consequently, all objects / sets which play the role ‘widget’ have the same shelf-life. Each set also has a ‘current-age’ parameter, which defines how *old* this set is? These two values are used to derive the age of any set, which is a comparator for the constraint. A code-age constraint is considered violated if one or more sets allocated to the order (or one of the child activities) has age more that the required age.

3.3 Conclusions

In this chapter we have presented our view, and modeling of the supply chain scheduling problem. We have also presented PODL, which is not only a problem description language, but also a problem specification paradigm. A complete example problem is given in Appendix A.