APPENDIX

AN OBJECT MODEL FOR SCHEDULING ALGORITHM IMPLEMENTATIONS (OMSAI)

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF ACRONYMS

BaB	Branch and Bound Class
HFS	Hybrid Flow Shop
HFSC	HFS Class
OMSAI	Object Model for Scheduling Algorithm Implementations
PMC	Parallel Machine Class
STL	Standard Template Libraries
TLC	Template List Class

APPENDIX

AN OBJECT MODEL FOR SCHEDULING ALGORITHM IMPLEMENTATIONS (OMSAI)

Abstract

Programming a large scheduling project on a computer is a complex and time-consuming task. In our project development in this thesis we used an object-oriented based model. This model has allowed us a better computer implementation and has reduced debugging and implementation times. We present an Object Model for Scheduling Algorithm Implementations (OMSAI). Such a model is composed of two sets of classes. The first set is composed of basis classes needed for scheduling algorithm implementations. These classes are elementary classes (like a job class and an arc class used to define a graph) and container classes which are mainly data structure classes. Container classes allow modeling shop floor configurations (in terms of number of machines and number of stages) as well as network structures. The second set of classes implements several algorithms. We present how to implement the general branch and bound algorithm and some network problems. The main objective here is to present a general framework for developing scheduling algorithms. One can then either use or improve this model for future implementations. One can simply draw one's inspiration for developing a specific algorithm.

1. INTRODUCTION

An object-oriented based model has been used for implementing the algorithms developed in this thesis. This model has reduced the debugging and implementation times and made it possible to develop the whole project in an incremental way.

We present a general Object Model for Scheduling Algorithm Implementations. The main objective here is to present the general framework within which the scheduling algorithms presented in this thesis were developed. One can then either use or improve the proposed model for future implementations, or simply draw one's inspiration from this model for developing a specific algorithm.

Object modeling is widely used in computing for general modeling or computer programming. See, for example, Meyer (1996), Rumbaugh et al., (1993) and Stroustrup (1995). The advantages of object oriented modeling and programming are enormous and do no longer need to be proved. Some of the main advantages are modularity and clarity, rapid prototyping and developing, ability of tackling large projects more easily and the possibility of reusing the complete, or part of the, model in different projects.

In broad terms, an object model can be seen as the definition of a set of classes of concepts or objects where each object may represent a concrete or an abstract concept in

the modeled system. Therefore a job, a machine, a sorting routine or a scheduling algorithm can be seen as objects in the case of scheduling projects. The object model defines these classes not only in terms of their contents and functionality but also in terms of the nature of their mutual relationships.

In section **2** we present the motivation for using object modeling for scheduling problems. In section **3** we present our Object Model for Scheduling Algorithm Implementations (OMSAI). In this section the model is presented by defining several sets of concept classes. In subsection **3.1** we present a description of the designed basis classes, necessary for our model. Needless to say that the definitions of these classes can be refined and/or extended for different projects. Subsection **3.2** presents several concept classes, which are used to define *four* algorithmic models. The first model is used for implementing such general scheduling solutions as upper and lower bound algorithms. The second model defines a general branch and bound model, which can be used and extended for implementing *Branch and Bound algorithms*. The last two models are network models, one for implementing the *longest path algorithm* and one for implementing *maximum flow algorithms*. In the remainder of this section we briefly recall some elementary definitions of the concepts used in object modeling. The reader who is familiar with object modeling can directly go to Section 2. For a detail study of object modeling one can refer to Meyer (1996), Rumbaugh et al., (1993) and Stroustrup (1995), and the references cited therein.

1.1 OBJECT CONCEPT

Object. An object is an independent *data structure* used to model and to represent a concept as well as its functionality in a system or a project. The object models the static and the dynamic part of the concept. An object is then defined by a set of proprieties or *attributes* describing its nature and/or its state. It is also defined by a set of methods or *member functions* allowing it to take such actions as updating its attributes or invoking other object functions. The set of member functions defines what is referred to as the *functionality* of the object. These member functions give the object the language or the ability to behave and interact with other objects in the modeled system.

In classical programming terms, an object attribute can be seen as a variable or an elaborated data structure of any type (i.e., a set of variables as an array, a list, a queue, etc.). The member functions of an object can be viewed as regular functions or routines in classical programming.

The main difference is that attributes and member functions are local to the object (like local variables in classical functions). Also, member functions are the object property and can not be used except by the object itself. Their impact occurs only in the object context (e.g. can only modify the object attributes). An object function must be invoked by specifying the object name. It has a direct effect on the object by either updating its attributes or invoking another function.

While the object attributes can easily be defined when designing an object, defining its functionality may not be an easy task. Indeed, the functionality of an object determines the limits or borders of the object in interfering with other object functionality.

Class of objects. An object is a concrete representation of an instance of an object class. The class represents the set of objects having the same attributes and functionality. Thus, an object class defines any concrete object or instance. Note that the instance definition in terms of attributes and member functions is made once and for all at its class level. Concrete objects or instances with attributes of different values can then be instantiated from the class defining their characteristics.

Aggregation and Specialization. In object modeling, aggregate or general concepts can be defined by using an aggregate view of the represented objects. An object class models such aggregate view. Specialized concepts can then be deduced from such aggregate class by refining the aggregate class definition. Such refinement is achieved by adding other attributes and/or functionality to the derived class, called subclass. One can then define different levels of aggregation by using a hierarchy of classes. Each class or super-class is refined in a set of subclasses. For example, one can define a production shop floor as a black box which definition can be refined (by specialization) as composed of a set of resources and a set of materials. The set of resources can then be refined as different kinds of machines and tools, etc.

Inheritance. Inheritance helps to model aggregation and specialization concepts by letting a class or a subclass inherit the attributes and functionality of its super-class without having to redefine them at the subclass level. Once we have defined a "*resource*" class as having an availability time attribute, we no longer need to define this attribute in the "*machine*" subclass. The attribute definition is simply inherited when stating that the machine is a resource. Furthermore, other attributes and functionality can be added at the subclass level so as to enrich the subclass definition. Also, functionality inherited from a class can be redefined in order to accommodate the functionality at the subclass level. The modeled system can then be represented by a graph of a hierarchy of classes, depicted by a tree or a set of trees. Instances can then be created from any level of the hierarchy of classes to represent a concrete object in the system, e.g., while a computer program is running.

Instantiation. Instantiation is the process of creating a concrete object or instance using its class definition, e.g. job#1 is instantiated from the "Job" class defining the jobs to be scheduled. The created instance has the same attributes as those defined and inherited by its class. Such attributes contain specific information on the instance. An instance is able to respond to any call of any function defined in, or inherited by, the class wherefrom it is instantiated. When called, a function is executed in the context of the instance receiving such call. For example, when job#1 is assigned to a machine and its starting time is known, a function is invoked to update its starting time attribute. Such function has been defined in the class from which job #1 is instantiated.

1.2 ADVANTAGES OF OBJECT MODELING

The main advantages of object modeling are modularity and clarity, rapid prototyping and developing, model reuse together with the ability of tackling large projects in a more convenient way. An important and obvious advantage, due to aggregate concept modeling, is the possibility of reusing the complete, or part of the, model for different projects. For example, one can use a set of predefined classes (like data structure classes) in different applications.

Modularity and Clarity. Since any element of a project is modeled as an object or a class, the final model is regarded as a set of objects linked hierarchically and/or by their possible mutual interactions. With such representation, we have a better view of the whole project in terms of its components and their possible interactions. One can then easily modify and add components for project extension purposes.

Rapid prototyping and developing. The object representation allows tackling small independent pieces of the project (object classes) instead of a frontal attack on the whole project. By doing so, one reduces the debugging time as well as the complexity of tackling the whole project in a straightforward manner.

Model reuse. The ability of reusing the same, or slightly modified, pieces of the model for different projects or for project extension purposes is one of the key advantages of object modeling. This is mainly due to aggregate concept modeling. Indeed, the functionality of each class is well defined and has some independence vis-à-vis the whole project. It is then easy to use a predefined class in other projects in its current definition or after minor adaptations.

2. OBJECT MODELING FOR SCHEDULING PROBLEMS

MOTIVATION

Production scheduling problems have widely been studied and their solutions always need computer implementations. Also, existing scheduling algorithms are implemented for testing purposes, which results in multi-implementation of the same solutions by many researchers and developers. Furthermore, because of the improvement in computing capabilities, new researches are tackling larger and more complex production scheduling problems.

Thus one can see that the same basic concept (e.g. a job or a machine), and the same basic scheduling algorithms may be used in different scheduling projects. As a matter of fact, some concepts and algorithms are so consistent that their modeling for multi-use is possible. All these factors, among others, emphasize the need for general models for scheduling algorithm implementations to tackle large and complex problems as well as for rapid project prototyping and development.

OBJECT CONCEPTS IN SCHEDULING

Scheduling is a matter of assignment or allocation of scarce resources. It might also be a matter of task or job sequencing on these resources. Such functions can be modeled using object concepts.

For example, in a shop floor, a set of resources can be grouped to represent a pool of capacity. One can then define a set of general characteristics of such pool of capacity which can be modeled using the shop structure and the operations used for scheduling. One can model each of these concepts by an object class. When implementing a specific scheduling solution one simply need to create an instance of the specific shop configuration.

The job concept can be regarded as having a minimum definition as the corresponding number (or name), the processing time and the completion time. Also, a specific elementary scheduling operation is used many times and in different projects. For example a resource or a machine is always assigned to a person, a job or a task. Once the processing of a job starts on some machine, this machine remains busy until the processing of the job is completed. The job starting time as well as the machine completion time might then be deduced once such assignment occurs.

What we are to model in this project are these structures and logic that are so general that, once modeled, they can be used every time needed, by simply selecting the corresponding part of the model.

In the remainder of this appendix we model some scheduling algorithm implementations using the object concept. The hybrid flowshop problem serves us as shop floor basis for our model.

3. AN OBJECT MODEL FOR SCHEDULING ALGORITHM IMPLEMENTATIONS

We now define a set of classes needed for scheduling algorithm implementations. These classes have been used in our project to implement the developed algorithms. Note that we present here only a general framework, not all classes are presented and some definition details are omitted.

For some presented classes we will also give comments on the class definition and its possible extension. The remainder of this section is organized as follows. Subsection **3.1** presents structural classes. We define there the basis classes needed for the final algorithms. These classes are mostly container classes (e.g., a list, an array, queue, etc). We present the job class, the machine class, how a machine class can be used to define a parallel-machine class and in the same way the hybrid flowshop class. Network or graph representations are widely used in scheduling problems and more generally in combinatorial optimization. We also present an object model for a graph representation of a schedule and how this graph class can be used in different algorithms. Subsection **3.2** presents some algorithmic classes. We present how upper and lower bound classes can be implemented. We also present object models for branch and bound, longest path and the maximum flow algorithm implementations. The choice of such algorithms is driven by our project needs. Subsection **3.3** presents the overall model.

3.1 STRUCTURAL CLASSES

The classes presented in this subsection are the basis classes for scheduling problems. They are more related to the scheduling problem structure than to a chosen algorithm. The HFS problem serves us as a basis for modeling, however one can select only the needed classes for a specific scheduling problem.

3.1.1 JOB CLASS

The job concept is not easy to model because its attributes and functionality depend on the structure of the production facility. For instances, in a multistage HFS production

facility with non-identical parallel machines, a job may have at each stage different processing times depending on the machine to which it is allocated. In this model we have chosen the HFS case with identical parallel machines as basis for our model. Such a model can always be extended for other HFS variants.

The job class is used to define job instances in a scheduling algorithm. In general, a job has a *name*, usually an integer value, to differentiate it from other jobs. The job also has other characteristics such as a *processing time*. We decided to model the job as an operation at some stage having a *release date*, a processing time and a *tail* (see the single stage subproblem in Chapter II, Section 5). So in the classical flowshop, the job shop and the HFS facility we have as many sets of jobs as stages. For illustrative proposes, some attributes and functions are presented though not used in our application.

Attributes

- *Name, release date, processing time, tail*
- *Starting time, completion time, waiting time, setup time, due date, makespan*

Member Functions

- *Functions for initializing and accessing all attributes.*
- *Compute completion time*
- *Compute makespan:* the job makespan is equal to its processing completion time plus its tail.

Comments

- A job makespan is equal to its processing completion time plus its tail.
- Note that if splitting and preemption are not allowed the completion time of a job is known when the starting time is known. Also, the completion time of a job may also indicate the release time of the job at the subsequent stage when overlapping is not allowed.
- If we want to model the case when the job has several processing times depending on the machine to which it is assigned (e.g., non-identical parallel machines), the processing time attribute is replaced by a set of possible processing times.

3.1.2 MACHINE CLASS

A machine, or resource, is used to process a task, a job, etc. Its assignment to a job results in the consumption of its available capacity, i.e. reduces its availability for certain amount of time in the case of a machine. A machine is defined by its *name* and the *nature of its capability*. A machine able of processing more than one job at a time has a *capacity* in terms of the maximum number of jobs that could be processed simultaneously. Each time a job is assigned to this machine this number is reduced by one.

In our case a machine processes one job at a time and we are only interested in knowing the set of jobs assigned to it as well as the completion, or the availability, time of the machine. We also define a *machine-makespan* attribute equal to the maximum over all assigned job makespans (see "Job" class). The machine-makespan value allows computing the makespan of the global schedule.

Since a specific machine is assigned to a specific job, a machine may have as attribute a set of assigned jobs. The definition of a job used here depends on the information we need to save for future algorithm needs. Such information could be a simple integer value

specifying the job name or any job elaborated definition as the one given in the "Job" class. The set of jobs could be an empty set if we are not interested in knowing which job is assigned to that machine.

Figure 6.1 gives two specialization of "Job" class, a job storing the initial date (i.e. "Job_1" class) and an "Assigned job" class. In the latter case we are only interested in knowing the starting and completion times. So the list of jobs assigned to a machine could be a list of jobs instantiated from "Job" class or from "Assigned job" class.

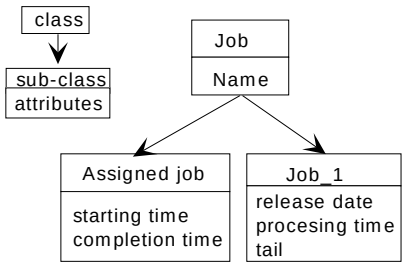

Figure 6.1: A hierarchical aggregate view of a job

We next give the complete "Machine" class definition

Attributes

- *Name, completion time, makespan, set of jobs*

Member Functions

- *Functions for initializing and accessing all attributes*
- *Putting a job on the machine*: this function adds a job to "*set of jobs"* and computes its starting time as well as the machine *completion* and *makespan* times. It takes as a parameter a job and updates the attribute "*set of jobs"*.

Comments

- One can create separate assignment and sequencing functions. The assignment function add a job to "*set of jobs"*, and the sequencing function fixes, according to some rule, the sequencing of the jobs on the machine. The latter function also updates the starting and completion times of the jobs and the machine.

3.1.3 TEMPLATE CONTAINER CLASSES

One can use any *template data structure class* (a list, an array, a queue, etc) to store and handle a set of objects. Such classes are referred to as *template container classes*. Standard template container classes are implemented in most object-programming languages as Standard Template Libraries (STL). See, for example, Glass and Schubert (1995) or any object programming language reference manual. STL is made up of a set of classes allowing one to handle and manipulate any container type (a list, an array, a queue, etc) of any object type (standard types like integers and strings, or elaborated object classes). So one can add, access or remove any element in a specific container regardless of the object type handled. One simply has to select the container needed and specify the contained object type. As in classical programming, one should choose the appropriate containers for the developed algorithm.

In our project, for instance, we use a Template List Class (TLC) defined in STL. To define "A job list" class, for example, we deduce a subclass from TLC, which we refine

by adding some specific sorting functions needed for our problem. Figure 6.2 gives three specializations of TLC: *"A job list"*, *" A machine list"* and *"BaB List-of-nodes"* used in the branch and bound algorithm, (See later Subsection 3.2.4). To work with a list instance we simply create an instance from "A job list" class by specifying the contained object class, i.e. a "Job" class.

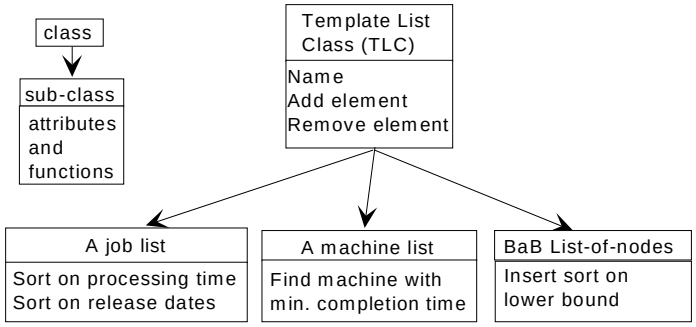

Figure 6.2: A hierarchical aggregate view of a list

3.1.4 PARALLEL MACHINE CLASS

A Parallel Machine Class (PMC) is defined by a *set of machines* (a container class), the *minimum completion time* and the *maximum makespan* over all machines.

One can define several assignment rules. A job can be assigned to an already designated machine, to the first available machine, i.e. with the minimum completion time, or using any other assignment rule. The set of machines is an indexed container, i.e. an array. Thus, the machine assignment to a job is done in constant time. Note that once a machine is assigned to a job both the machine and the parallel machine attributes are modified.

Attributes

- *Name or stage, set of machines, available machine, makespan*

Member Functions

- *Functions for initializing and accessing all attributes*
- *Select a machine*: this function selects a specific machine according to some criteria
- *Add a job to a specific machine*: this function takes a job and a machine number as parameters. The specified machine is assigned to the specified job
- *Add a job to the available machine*: this function takes a job as a parameter and assigns the available machine to it
- *Schedule the stage*: This function can apply a specific scheduling algorithm to the parallel machines. It returns a set of values corresponding to the completion times of the jobs at the scheduled stage.

Comments

- Both last mentioned functions know which machine and which job to be assigned to. They simply invoke the '*Putting a job on the machine*' function defined in the "Machine" class. When applied, this function returns the new completion time of the machine, which is used to update the attributes 'available machine' and 'makespan' at the parallel machine level.

- Note also that any request to a machine instance must always pass via the corresponding PMC instance since a machine is always referenced at the corresponding stage, i.e. corresponds to a single stage problem.
- Most of the functions defined in PMC will always invoke a machine function and apply the impact of such operation to the PMC instance attributes.

3.1.5 HYBRID FLOW SHOP CLASS

A hybrid flow shop is composed of a set of stages of parallel machines (a container class). A Hybrid Flow Shop Class (HFSC) allows the handling of the HFS scheduling problem. Although a specific algorithm class may be used for effecting the specific scheduling, the HFSC is used to create the HFS designed structure in terms of number of stages and their corresponding number of parallel machines. Actually, the HFSC instance is used as an attribute of an elaborated algorithm class. However, basic algorithms can be implemented as member functions of HFSC.

Attributes

- *Set of stages, current stage, makespan*

Member Functions

- *Functions for initializing and accessing all attributes*
- *Create HFS*: this function creates an HFS structure according to the given parameters: the number of stages and the number of machines per stage
- *Schedule the HFS:* this function is used when a sequential scheduling is carried out. It simply applies successive stages scheduling by invoking next functions
- *Schedule a stage*: This function invokes the "*Schedule the stage*" implemented in the parallel machine class, which effect the scheduling of the specified stage. From PMC, it gets a set of values corresponding to the completion times of the jobs at the scheduled stage and invokes the next function
- *Update release dates*: this function takes the jobs completion times at a stage, computes their release dates at the subsequent stage, and does the necessary updating
- *Schedule with A job list*: This function is a special case of '*Schedule the HFS*' function using an ordered job list to schedule the HFS. Another algorithm class carries out the ordering of the jobs according to some criterion.

In Chapter III and IV we used the graph representation for scheduling and for solving the maximum flow problem. The Graph class is used to handle the graph creation and exploitation. A set of arcs and a set of nodes define the graph. We first define the "Node" class and the "Arc" class.

3.1.6 GRAPH NODE CLASS

The "Node" class models the node concept used in a graph. A node, here, is defined in terms of its links in the graph so one can easily access the adjacent nodes.

Attributes

- *Name, a value, set of successor arcs, set of predecessor arcs*

Member Functions

- *Functions for initializing and accessing all attributes*

- *Insert a successor:* Add a successor arc to this node
- *Insert a predecessor:* Add a predecessor arc to this node
- *Scan adjacent arcs/nodes*: this function scans adjacent arcs/nodes, using the sets of successor and predecessor arcs, to select an arc or a node with a specific value

Comments

- The '*Scan adjacent arcs/nodes'* function can be split into two functions, '*Scan successor nodes' and 'Scan predecessor nodes'* to allow separate forward and backward moving in the graph.
- One can create different sets of successor arcs and different sets of predecessor arcs to facilitate the addition and the removal of the sets if these operations are needed. In the graph representation of a schedule we use two types of arcs, conjunctive and disjunctive arcs. In the single stage based heuristics (see Chapter III) and when rescheduling the critical stages, each time we obtain a new schedule of the single stage we replace the old one in the graph by removing the corresponding disjunctive arcs and add the new ones.

3.1.7 GRAPH ARC CLASS

The "Arc" class models the arc concept used in a graph. An arc is used to link two nodes and usually has a *weight* or a capacity and an *orientation*. If the weight corresponds to a capacity other attributes might be needed like the *unused capacity* and the *capacity cost*.

Attributes

- *Capacity, unused capacity, left-hand side node (the immediate predecessor) , right-hand side node (the immediate successor)*

Member Functions

- *Functions for initializing and accessing all attributes*
- *Update capacity/unused capacity*

3.1.8 GRAPH CLASS

This class is used to implement the basis for creating and exploring a graph. A graph is composed of a set of nodes and a set of arcs. In our model a "*graph*" can be regarded as defined by a set of nodes where each node is defined by two sets of arcs, the incoming and outcoming arcs, linking the node to the adjacent nodes. The node class already encompasses the arc definition and since an arc is defined as linking two nodes, we only need to know a specific node or a specific arc (linking two nodes) to explore the entire graph.

Attributes

- *Set of nodes, current node, current arc*

Member Functions

- *Functions for initializing and accessing all attributes*
- *Build a graph*: this function builds a graph either by reading the graph structure from an external file or by translating the given information into nodes and arcs linking these nodes.
- *Insert node*: add a new node into Set of nodes (the graph).

- *Insert arc*: link two nodes of the graph by an arc by invoking the *"Insert a successor" and/or "Insert a predecessor"* functions of the two linked nodes.
- *Move to node*: this function changes the current node to the designated one. It is used for moving in the graph and usually does some calculation involving the nodes and/or arcs attributes.

Comments

- The functions defined here are mainly used for building and exploring the graph. Functions for specific graph algorithms are implemented in refined subclasses. Figure 6.3 shows two subclasses, used in our project, and derived from the graph class: a graph representation of a schedule and a graph class used for solving the maximum flow problem.

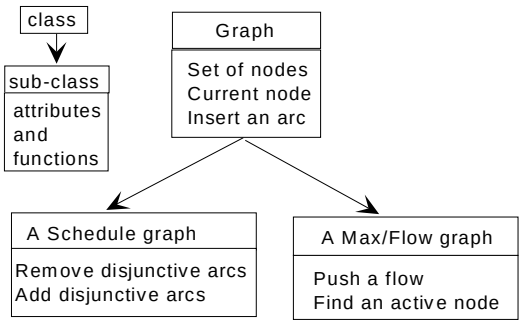

Figure 6.3: A hierarchical aggregate view of a graph

- Since this class is a container class it should be implemented as a template class. So one can define different node and arc types depending on the problem specificity.

3.2. ALGORITHMIC CLASSES

3.2.1 UPPER AND LOWER BOUND CLASSES

We suggest here that a major specific class or subclass should define each specific upper or lower bound algorithm, so one can always use this algorithm in different projects. A typical upper or lower bound class requires the scheduling problem data including the shop structure in terms of number of stages and number of machines by stage. This information is used to build the HFS instance derived from HFSC class, see Subsection 3.1.5. A set of functions or subclasses might need to be developed to implement the specific upper or lower bound algorithm. In our project a class called '*Schedule HFS*' is defined for scheduling the HFS one stage at a time (see Chapter III: Upper Bounds). This class creates the HFS structure and handles the problem data. From this class different subclasses are defined for developing different classes of algorithms. Figure 6.4 gives an overview of three classes derived from '*Schedule HFS*' class.

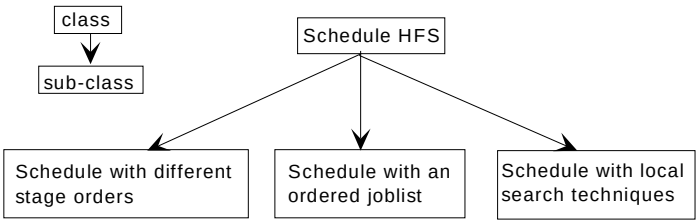

Figure 6.4: An overview of three classes derived from 'Schedule HFS' class

3.2.2 BRANCH AND BOUND ALGORITHM

Branch and bound algorithms are widely used in combinatorial optimization. The necessary elements composing a branch and bound algorithm are: a branch and bound tree made up of a set of nodes, node selection rules, branching rules, upper and lower bounding algorithms. In the following subsections we present three basis classes for implementing general branch and bound algorithms. The *node class* handles the subproblem specific data. A list of *active nodes* is used to store the created and not yet explored nodes. *"BaB" class* implements the usual operations carried out in a branch and bound algorithm like node selection and branching operations. Note that upper and lower bound algorithms should be implemented in different external classes.

3.2.3 BRANCH AND BOUND ALGORITHM CLASS

The Branch and Bound algorithm class (BaB) holds the original problem description, the best known solution as well as all non-explored derived subproblems (List-of-nodes), (see later "BaB node" and "BaB list of nodes" Classes). BaB class carries out the branching operation by selecting a branching node from List-of-nodes, creating nodes or subproblems, and storing these new nodes in List-of-nodes.

Attributes

- *Problem instance, List-of-nodes, upper bounds, lower bounds, stopping times, number of nodes, etc.*

Member Functions

- *Functions for initializing and accessing all attributes*
- *Start*: this function reads the problem data, creates the root node and asks the root node to compute its corresponding lower and/or upper bound (see later "BaB node" Class). It stores the root node in List-of-nodes and invokes the "*Branch and Bound*" function.
- *Branch and Bound*: this function implements the core of the algorithm. It iterates until List-of-nodes is empty or the stopping time has been reached. At each iteration, it selects a branching node from List-of-nodes. Recall that the way List-of-nodes is organized determines the order in which the branch and bound tree is explored (see later "BaB List-of-nodes" Class). This function may also ask the selected node to compute its corresponding lower and/or upper bound (see later "BaB node" Class). Eventually, it checks if the node is worth exploring and if so invokes the next function: '*Branching operation'*.
- *Branching operation*: This function is separated from '*Branch and Bound'* function because it is problem dependent and needs to be implemented for each specific problem. It creates the subproblems or nodes from the selected node (see Create node" in "BaB node" class) . The new created nodes are asked to compute their respective lower and/or upper bound and are stored in List-of-nodes.

Comments

- The branching operation can be implemented as a separate class. For branching, an instance is created by "BaB". It is supplied by the selected node and after processing the separation operation returns a (new created) list of nodes to BaB.
- Someone interested in developing a branch and bound algorithm needs only to develop the classes and functions specific to the problem solved. These are a

'Node' class, upper and lower bound algorithm classes, and the '*Branching operation*' class or function.

3.2.4 BaB node Class

Although the content of a node is problem dependent general attributes can be defined, like upper and lower bound values. Separate upper and lower bound algorithm classes carry out their computations. Someone developing a branch and bound algorithm needs just to create a subclass from "BaB node" class to which specific attributes and functions are added.

The node holds the information on the corresponding specific subproblem. This information is used to compute the node's upper and lower bounds. This information could be a complete description of the subproblem or just some specifications making the node different from the other nodes. For example, in the interval method presented in Chapter V we used a list-of-jobs to save information on the jobs release dates, due dates and tails, which were modified by the branching scheme. So one should define a class to represent the subproblem specification whose instance is an attribute in class "BaB Node" class.

Attributes

- *Name, upper bound, lower bound, subproblem instance*

Member Functions

- *Functions for initializing and accessing all attributes*
- *Create node*: this function builds the node by creating an instance defining the new subproblem which is stored in the '*subproblem instance*' attribute. This function knows exactly how to create the specific corresponding subproblem.
- *Compute upper bound*: this function creates an instance from the "*upper bound*" class and transmits the *subproblem* as a parameter. The returned upper bound value is stored in the "*upper bound"* attribute.
- *Compute lower bound*: this function creates an instance from the "*lower bound"* class and transmits the *subproblem* as a parameter. The return lower bound value is stored in the "*lower bound"* attribute.

Comments

- Attribute and member functions related to a specific problem are better defined in subclasses as shown in Figure 6.5. The nodes needed for the specific problem are instantiated from the corresponding subclass.

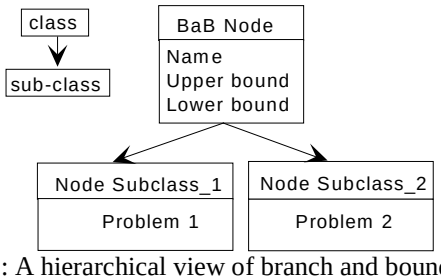

Figure 6.5: A hierarchical view of branch and bound nodes

3.2.5 BaB List-of-nodes Class

Class "List-of-nodes" is a subclass, derived from the "Template List Class" as shown in Figure 6.2. It holds the created and not yet explored nodes. The way the list is organized determines the order in which the branch and bound tree is explored. A *stack* results in a *depth-first* search whereas a *queue* results in a *breadth-first* search. In our project this list is sorted in a non-decreasing order of node lower bounds.

Attributes

- *List-of-nodes*

Member Functions

- *Insert/Sort a node:* this function insert by sorting a node in "list of nodes".
- *Remove first node:* this function removes and returns the node with the smallest lower bound.

Comments

- The list is organized by maintaining a sorted list of nodes using a *heap*, such that the insert or remove operation is done in O(log n), where n is the number of nodes in the List-of-nodes.

3.2.6 Longest Path Algorithm Class

This class is used to compute the longest path in an acyclic directed graph. Note that in the same way a shortest path class can be defined or derived from this class by redefining the comparison function used.

This class is a subclass of "Graph" class, see Subsection 3.1.8. The "*Longest path value*" attribute stores the computed longest path value from a specific node to another specific node in the graph. However, this value needs to be updated each time the graph is modified, i.e. an arc or a node is added, or when the distance or weight of an arc is modified. When found the longest path might be stored in a set of arcs, '*longest path*', which also needs updating if the graph is modified.

Attributes

- *Longest path value, longest path*

Member Functions

- *Functions for initializing and accessing all attributes.*
- *Longest path from node to node*: this function computes the longest path from any node in the graph to any other node and stores the corresponding value and path.

Comments

- This class has been used in the graph representation of a schedule to compute jobs release dates and tails.
- This class is implemented as a subclass of the "*Graph*" class because in graph problems we are not always interested in computing the longest path.

3.2.7 Maximum Flow Algorithm Class

This class is used to implement algorithms for solving the maximum flow problem. It is a subclass of the "Graph" class. It implements the basic operations needed for solving the maximum flow problem, like pushing a flow from node to node and computing the exact

distance labels. See Chapter IV for a description of the maximum flow problem. In the "*Preflow algorithm*", for example, we need to know the distance label and the excess flow of each node. For that purpose, a new subclass "A graph node for Max/Flow" is derived from the "Graph node" to which these two attributes are added. The "Max/Flow algorithm" instance is instantiated using the "Graph node for Max/Flow" class. All attributes are inherited from "*Graph*" class.

Member Functions

- *Functions for initializing and accessing all attributes.*
- *Compute distance labels:* this function computes the exact distance label for all nodes.
- *Update exact distance label:* this function computes the exact distance label for a selected node.
- *Push:* this function pushes a flow from a specified node to another specified node.
- *Find max/flow*: this function finds the maximum flow using the Preflow/push algorithm presented in Chapter IV, Section 7. It uses the '*Update exact distance label'* and *'Push'* routines to carry out single operations.

3.3 THE OVERALL MODEL

Figure 6.6 presents an overview of the overall model. So for the sake of clarity, this figure depicts only some illustrative classes.

The model is made up of several sets of classes linked by three types of links. A class could be a (specialized) subclass derived from an aggregate or super-class (a *hairline*). A class could be used as an attribute in another class (a *dash-line*). Or a class could be defined as a set of classes (*bold-line*). Classes defined by a set of classes are *container classes*. They are mainly subclasses of *template classes*. We hereafter give some remarks concerning some classes and their links in Figure 6.6.

All container classes are subclasses of "Template containers" class. Container classes use an elementary object class. Such classes are: the "Job" class, the "Machine" class, the "Node" class and the "Arc" class. Note that subclasses are defined from these elementary classes. (e.g. The "Assigned job" class is a subclass of the "Job" class).

Shop floor configurations

- The "Machine" class is the basis class used to define different shop floor classes. It is first used to define a "Set of machines" class regardless of the shop configuration. The one-machine problem is a subclass of the "Machine" class.
- The "Set of machines" class can be refined to define a "Parallel machine" class, a "Flowshop" class or a "Jobshop" class.
- From the "Parallel machine" class a "Set of parallel of machines" class is defined. The latter is refined to define a "Hybrid flowshop" class and a "Hybrid jobshop" class.

Jobs data

- We use two specialized job classes, the "Job_1" class defining the initial problem data, and the "Assigned job" class defining scheduling information like starting and completion times.

- Each job class is used to define a container class. "List of jobs" class is, in general, used to handle jobs and "List of assigned jobs" is used to handle the already scheduled jobs.

Shop floor

- "Configuration" class provides general information (coming from an external file) on the carried out *scheduling numerical tests*. It specifies the shop floor configuration in terms of the number of stages, number of machines and number of jobs. It also specifies the file name of the problem data. Eventually, it specifies the set of algorithms to be tested.
- "Shop floor" class has three attributes, an instance of "Configuration" class, an instance of "List of jobs" class and an instance of the shop floor considered for scheduling, i.e. "Hybrid flow shop" class.
- Jobs data might be created (according to some selected distribution) by the "Data generator" class and stored in an external file for future use.

Scheduling Algorithm classes

- The "Scheduling algorithms" class has a "shop floor" class instance as an attribute.
- "Scheduling algorithms" is refined to define specific algorithmic classes, the "Lower bound algorithm" class, the "Upper bound algorithm" class and the "BaB" class. Specific algorithms can be derived from these classes.

Branch and bound classes

- The "BaB" class can be used to define different branch and bound algorithms. It has as attributes an instance of "Lower bound algorithm" class and an instance of "Upper bound algorithm" class.
- The "BaB i" (i=1,2,…) is a specialized "BaB" class for a specific problem.
- So, in order to define a specific algorithm one should define a subclass of "BaB Node" class and subclasses of upper and lower bound classes.

Implementing an algorithm

- In order to implement an algorithm for the HFS scheduling problem (for example) one just need to create an instance of the corresponding class, i.e. a lower bound algorithm, an upper bound algorithm or a branch and bound algorithm class.
- The algorithm instance creates in turn an instance of "Shop floor" class. Which in turn creates the three composing instances, a "Hybrid flowshop" instance, a "List of jobs" instance and a "Configuration" instance. Eventually, the algorithm instance, based on this information, processes the corresponding algorithm.

4. CONCLUSION

Programming a large number of scheduling algorithms on a computer is a complex and time-consuming task. In this appendix we have presented an Object Model for Scheduling Algorithm Implementations. Such model has been used for implementing the algorithms developed in this thesis. It has reduced the debugging and implementation times and made it possible to develop the whole project in an incremental way.

The model is presented as definitions of object classes. Two sets of classes compose the general model. The first set is made up of basis classes needed for scheduling algorithm implementations. These classes are elementary (e.g. the “Job” class and the “Arc” class) and container classes. Container classes allow to model shop floor configurations (in terms of number of machines and number of stages) and a network structure. The second set of classes implements several algorithms. We presented how to implement the general branch and bound algorithm and some network problems.

The main objective was to present a general framework for developing scheduling algorithms. One can then either use or improve this model for future scheduling algorithm implementations. Also, one can simply draw one's inspiration from this model for specific algorithm implementations.

REFERENCES

1. Glass G. and Schubert Brett, (1995). The STL <Primer>, Prentice Hall, 1995.
2. Meyer B., (1996). Object Oriented Software Construction, Prentice Hall, 1996.
3. Rumbaugh J. M, W. Blaha, F. Eddy Premerlani, W. Lorensen, (1993). Object Oriented Modeling and Design, 1993, Prentice Hall.
4. Stroustrup B., (1995). The C++ Programming Language, Addison Wesley, 1995.

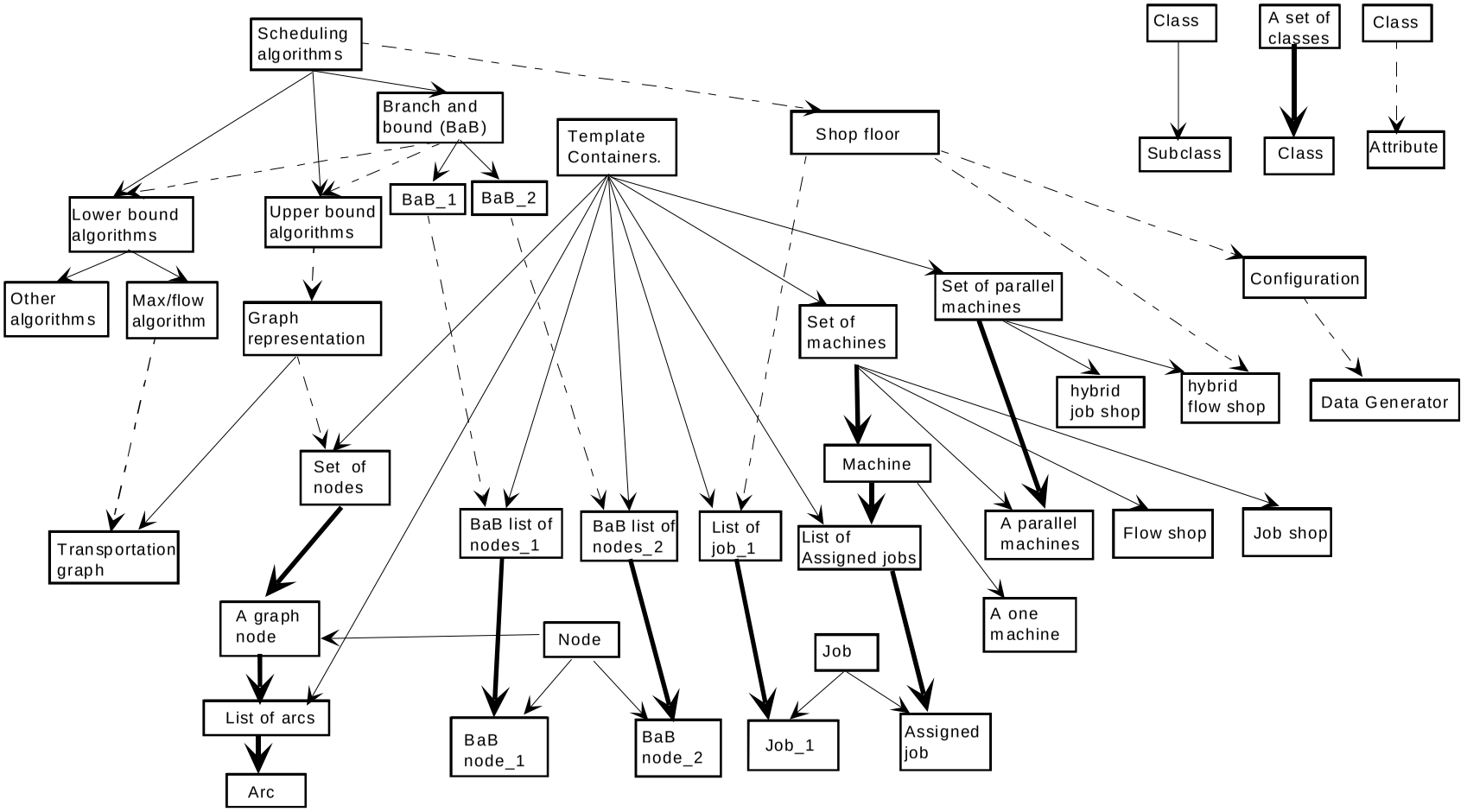

Figure 6.6: An Object Model for Scheduling Algorithm Implementations (OMSAI)