On the Design and Development of Object-oriented Scheduling Systems

Michael Pinedo
Leonard N. Stern School of Business
New York University

Benjamin P.-C. Yen
Department of Industrial Engineering and Operations Reseach
Columbia University

1997

Working Paper Series
Stern #IS-97-25

Annals of Operations Research 70(1997)359–378

On the design and development of object-oriented scheduling systems

Michael Pinedo and Benjamin P.-C. Yen*

Department of Industrial Engineering and Operations Research, Columbia University, New York, NY 10027, USA

In this paper, we describe the architecture of an object-oriented scheduling system. First, a mathematical framework is presented that is based on set theory and graph theory. Then a number of basic as well as more specialized methods are defined which can be applied on the entities of any decision support system. The principal objects of a scheduling system are defined, as well as the methods specifically designed for the manipulation of the schedules. The object base design, the schedule generator design and the user interface design are then discussed in detail.

1. Introduction

During the last two decades, a large number of scheduling systems have been developed in industry and academia. Some of these systems are application-specific, while others are generic. In the implementation process, application-specific systems usually fare somewhat better than generic systems that are customized. However, the application-specific systems are often difficult to modify and adapt to changing environments. Generic systems are usually somewhat better designed and more modular. However, any type of customization of such systems typically leads to significant investments. A description of the basic architecture and the main components of a typical scheduling system are presented in figure 1.

Considering the experience of the last two decades, it appears useful to provide guidelines that facilitate and standardize the design and the development of scheduling systems. In this paper, an attempt is made to provide guidelines as well as tools for the development of scheduling systems. The design presented here is in every aspect object-oriented.

There are many advantages in following an object-oriented design approach in the development of a scheduling system. First, the design is modular, which makes maintenance and modification of the system relatively easy. Second, large segments of the code are reusable. This implies that two scheduling systems that are substantially different still may share a significant amount of code. Third, the designer thinks

*Present address: Department of Industrial Engineering and Engineering Management, Hong Kong University of Science and Technology, Hong Kong.

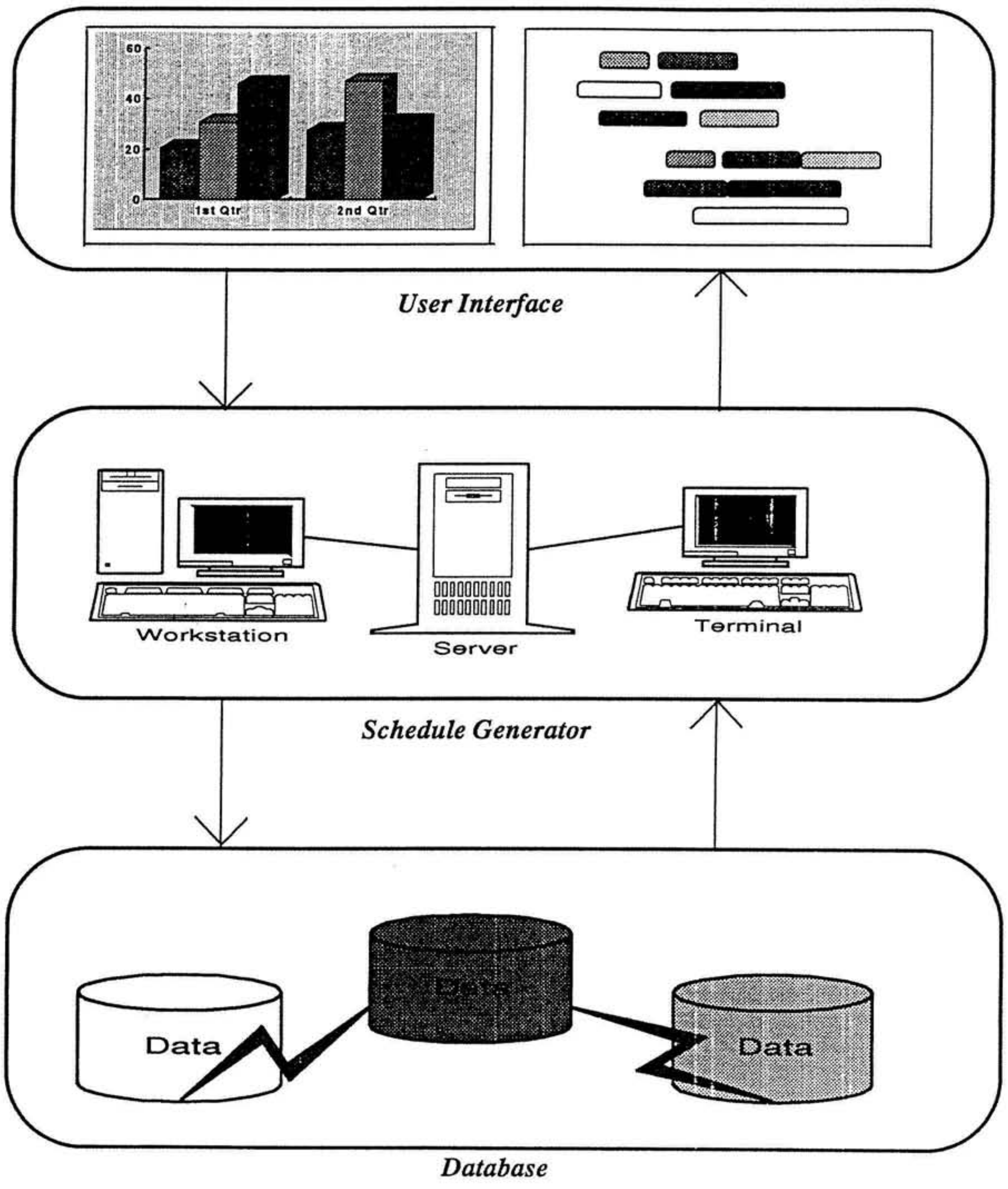


Figure 1. Components of a scheduling system.

in terms of the behavior of objects, not in low-level detail. In other words, the object-oriented design approach can speed up the design process and separate the design process from the implementation process.

Object-oriented systems are usually designed around two basic entities, namely *objects* and *methods* (see [1,5,6,15]).

Objects usually refer to various types of entities or concepts. The most obvious ones are jobs and machines. However, a schedule is also an object and so are user interface widgets, such as buttons, menus and canvases. There are two basic kinds of relationships between object types, namely the *is-a* relationship and the *has-a* relationship. According to an is-a relationship, an object type is a special case of another object type. According to a has-a relationship, an object type may consist of other object types. Objects usually carry along static information, also referred to as attributes, and dynamic information, usually referred to as the state of the object. An object may have a number of attributes which are descriptors associated with the object. An object may be in any one of a number of states. For example, a machine may be *working*, *off*, or *broken down*. Changes in the state of an object are referred to as *events*.

A method is implemented in a system by means of one or more operators. Operators are used to manipulate the attributes corresponding to objects and may result in changes of object states, i.e., events. On the other hand, events may trigger operators as well. The sequence of states of the different objects can be described by a state-transition or event diagram. Such an event diagram may describe the links between operators and events. An operator may be regarded as the way in which a method is implemented in the software. Any given operator may be part of several methods. Some methods may be very basic and can be used for simple manipulations of objects, e.g., a pairwise interchange of two jobs in a schedule. Some methods may be very sophisticated or high level. They may be intricate heuristics for application on given sets of jobs (objects) in given machine environments (also objects). The application of a method to an object usually triggers an *event*.

The application of a method to an object may cause information to be transmitted from one object to another. In what follows, such information is referred to as a message. Messages represent information (or content) that are transmitted from one object (for example, a schedule) via a method to another object (for example, a user interface display). A message may consist of simple attributes or of an entire object. Messages are transmitted at the times that events occur (caused by the application of methods to objects). Messages have also been referred to in the literature as memos. The transmission of messages from one object to another can be described by a transition event diagram. Such transmissions require the specification of protocols. However, the specification of protocols are at a level of detail of system design that we will not address in this paper.

Figure 2 depicts the interplay between objects, methods and messages. In order to see the sometimes complicated relationships between objects, methods and messages, consider the following example of a somewhat specialized object, a so-called *agent* object. This very specialized object usually contains other objects, e.g., machines, jobs and so on, as well as one or more methods. An input message into such an object is the states of one or more objects, while the output message may be the states of the same objects or other objects. An agent object usually performs a decision-making function.

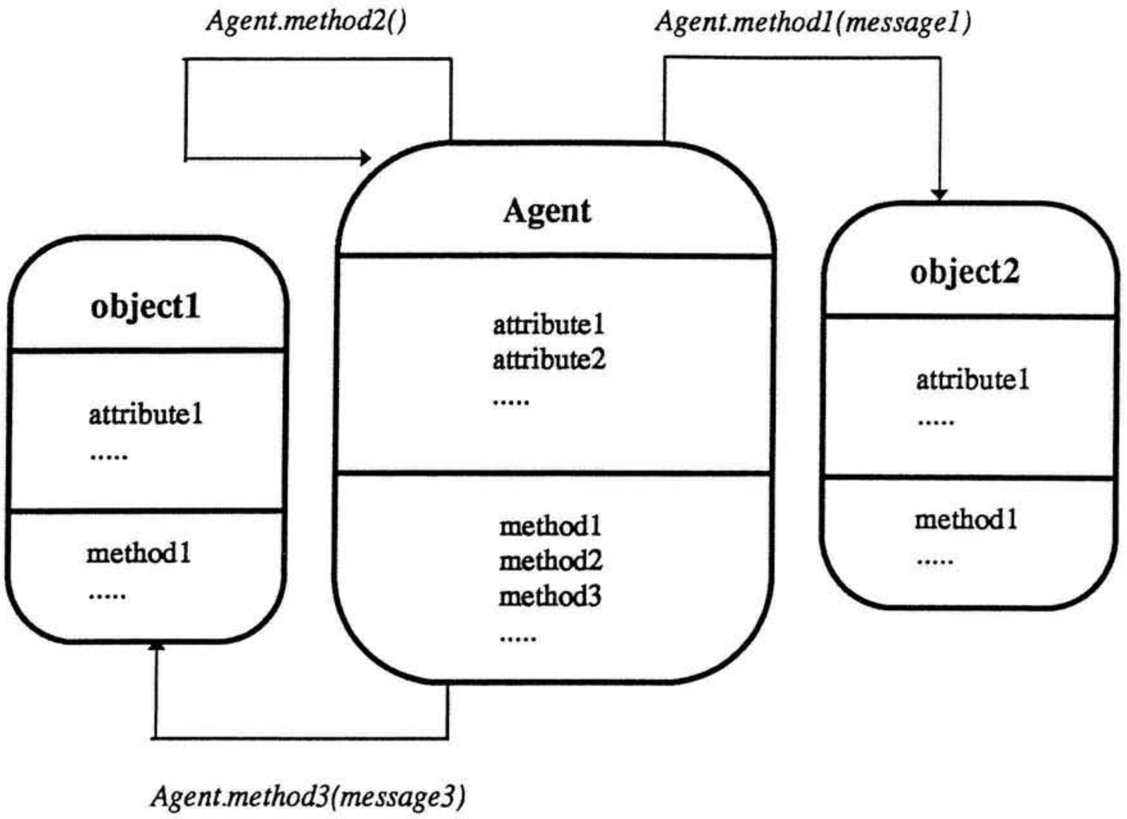


Figure 2. Agent object.

A scheduling system may be object-oriented in its conceptual design and/or in its development. A scheduling system is object-oriented in its conceptual design if the design of the system is object-oriented throughout. The design being object-oriented throughout implies that every concept used and every functionality of the system is either an object or a method of an object (whether it is in the data or knowledge base, the algorithm library, the schedule generator or the user interfaces). Even the largest modules within the system are objects, including the algorithm library and the user interface modules. A scheduling system is object-oriented in its development if only the more detailed design aspects are object-oriented and the code is based on a programming language with object-oriented extensions, e.g., C++.

Many scheduling systems developed in the past have object-oriented aspects and many systems tend to be object-oriented in their development. A number of these systems also have conceptual design aspects that are object oriented (especially with regard to the design of the respective databases and knowledge bases, see, for example, Collinot et al. [2], Fox and Smith [3], Smith [10], and Smith et al. [11]). Some of these systems relied on inference engines for the generation of feasible schedules and other systems were constraint based and relied on constraint propagation algorithms and search; these systems did not have schedule generators that perform very elaborate optimization. One of the exceptions is the schedule generator designed by Sauer [9], which is somewhat more elaborate than other scheduling engines. Also, a system design proposed by Smith and Lassila [12] extends the modular philosophy for scheduling systems farther than any one of the previous systems' designs.

Many systems developed in the past are not designed throughout according to an object-oriented philosophy. Some of the aspects that are typically not object-oriented include:

- the design of the schedule generators,
- the design of the user interfaces, and
- the incorporation of precedence, routing and layout constraints.

Few of the schedule generators developed in the past have extensive libraries of algorithms that are easily reconfigurable and may benefit from a modular object-oriented design. An object-oriented design would require a detailed specification of operators and methods. Since most scheduling environments would benefit from strongly interactive optimization, schedule generators have to be strongly linked with interfaces that allow schedulers to manipulate schedules manually as well. Still, object-oriented design has not yet had a major impact on the design of user interfaces of scheduling systems either. The precedence constraints, the routing constraints, and the machine layout constraints are often represented by rules in a knowledge base and an inference engine must generate a schedule satisfying all the rules. However, precedence, routing and layout constraints can be modeled conceptually easily using graph and tree objects, and these objects can then be used by an object-oriented schedule generator.

The purpose of this paper is the following. It describes a conceptual design for scheduling systems that is object-oriented throughout. The object-oriented design extends to *all* modules, including schedule generator modules and user interfaces. In other words, object-oriented system design is considered while regarding the representation of the scheduling problem, the mechanism for solving the problem (which is based on the representation), and the structure of the system supporting the mechanism as one entity.

This paper is organized as follows. In section 2, we present some elementary set theory and graph theory which provides a framework that is useful for the formulation of scheduling problems. In section 3, we describe some basic methods for the manipulation of the entities described in section 2. The concepts described in sections 2 and 3 are applicable to any kind of decision support system; it does not necessarily have to be a scheduling system. Section 4 focuses on the object-oriented modeling of scheduling problems. This frame of modeling is geared towards the development of the software necessary for dealing with the development of scheduling systems. Section 5 describes the object base design and section 6 deals with schedule manipulation methods. Section 7 discusses design aspects of schedule generators and section 8 goes into the design of user interfaces. The last section discusses implementation issues.

2. Preliminaries: Set theory and graph theory

Before venturing into object-oriented modeling, it is useful to review some very elementary set theory and graph theory.

A set refers to a collection of distinct entities. These entities are usually referred to as the elements of the set. A set can also have sets as elements. Given two sets S_1 and S_2, set S_1 is a subset of set S_2 if every element of S_1 is also an element of S_2. The notions of the union of two sets and the intersection of two sets are also well-known and therefore not defined here.

An ordered pair of objects is a pair of objects arranged in a fixed order. We use the notation (a, b) to denote the ordered pair in which the first object is a and the second object is b. The ordered pair (a, b) is not identical to the ordered pair (b, a).

A graph is usually defined as an ordered pair $G = (V, E)$, where V is a set and E is a binary relation on V. The elements in V are called the vertices and the ordered pairs in E are referred to as the edges of the directed graph. The concepts of graph theory are used when dealing with the precedence constraints, the routing constraints and the layout of the machine environment.

In a directed graph, a path is a sequence of edges $(e_{i1}, e_{i2}, \ldots, e_{ik})$, in such a way that the terminal vertex of edge e_{ij} coincides with the initial vertex of edge $e_{i,j+1}$. A path is said to be acyclic if it does not include the same edge twice. A tree is defined to be a connected graph with no cycles.

Precedence constraints are acyclic, while routing constraints may have cycles.

3. Basic methods

There are two types of methods that can be applied to objects associated with sets and graphs. The two types are the primitive methods and the functional methods.

The primitive methods are the most basic methods that can be applied to any type of object. The functional methods are object dependent and may be extendable. We consider the following five primitive methods:

(i)	construct:	create an instance.
(ii)	destruct:	destroy an instance.
(iii)	copy:	create a new instance identical to a given instance.
(iv)	assign:	assign a value to an attribute.
(v)	retrieve:	find an instance with a given attribute.

These five basic methods can be applied to objects that correspond to elements, sets, as well as graphs. There are many functional methods. In what follows, we present only a limited sample. For a more complete list, see Yen [14].

(i)	nullify:	set the default value or delete an element.
(ii)	include:	append an element to an already existing set.

(iii)	exclude:	remove an element from a set.
(iv)	compare:	compare two elements or sets and check whether they are equal or one includes the other.
(v)	merge:	combine two sets to form a new one.
(vi)	search:	search for a given attribute or value.
(vii)	sort:	sort in order of a certain attribute or value.
(viii)	min (max):	find among a number of objects the one with the minimum (maximum) value of a certain attribute.
(ix)	addnode:	add a node to a given graph.
(x)	deletenode:	delete a node from a given graph.

Some of these functional methods can only be applied to sets, whereas others can only be applied to graphs. Some can be applied to sets as well as graphs.

For example, the compare method can be applied to any object. It can show whether a set is a subset of another set and also whether a graph is a subgraph of another graph.

Applying the sort method on a set puts all the elements of the set in a given order. At times it may be useful to use arrays or lists instead of sets.

4. Object-oriented scheduling models

In object-oriented modeling of scheduling problems, it is necessary to determine what the most basic elements are in the given environment. At first sight, the most basic elements are the jobs, the machines and other resources such as tools. Actually, there may be even more basic elements than jobs. A job usually consists of a number of operations that have to be done on the various machines. An operation may be regarded as an element as well. In environments where preemptions are allowed, there is an element that is even smaller than an operation, namely an operation segment. The smallest element in a preemptive setting is an operation segment with a duration that is equal to the smallest time unit used in the scheduling system.

The following notation is used in what follows: an m refers to a single machine and an M refers to a set of machines. A j refers to a single job and a J refers to a set of jobs. A t refers to a point in time and an ordered pair $T = (t_1, t_2)$ refers to a time interval. An operation or operation segment o is an ordered triple (m, j, T). A set of operations or operation segments is denoted by O.

Graph theoretic concepts are often used in the modeling of scheduling problems. First, the machine layout of a given machine environment indicates the routes the jobs have to follow through the machine park. The machine layout may be depicted easily by a directed graph and depends only on information regarding the machines. Second, the precedence constraints on the various different jobs indicate the sequence in which operations from *different* jobs are to be processed. These precedence constraints can be represented by a directed graph as well. They depend only on information regarding the jobs. Third, the routing constraints with regard to a particular job indicate the route the job has to follow through the machine environment. As such, they indicate in what sequence and on which machines the various different operations of the given job have to be processed. These routing constraints can be represented easily by a directed graph. Often, however, routing constraints are specified by a routing table. This form of information is strongly associated with, but not completely identical to, the machine layout. Of course, consistency is required between machine layout and routing constraints. However, in some cases it is preferable to describe the job routing with a machine layout graph (especially when all jobs have the same route), and in some cases it is preferable to describe the job routings through routing constraints (especially when jobs have many different routes).

In order to illustrate the graph-theoretic notation, we present the notation for a machine layout where a single machine, say m_1, feeds into two machines in parallel, say machines m_2 and m_3. The three vertices of the graph are the three machines m_1, m_2 and m_3. The set of vertices is denoted by V_m and is $\{m_1, m_2, m_3\}$. There are two vertex subsets SV_m, namely $\{m_1\}$ and $\{m_2, m_3\}$. The set of all vertex subsets is denoted by V_l and is $\{\{m_1\}, \{m_2, m_3\}\}$. There is a single edge, namely $(\{m_1\}, \{m_2, m_3\})$. Let the set of all edge subsets be denoted by E_l. There is, clearly, only a single edge subset. The entire layout graph may be denoted by $G_l = (V_l, E_l)$.

The time is an important variable. An ordered pair of two points in time is referred to as a time interval. A time interval is a basic element as well.

Different types of schedules can be defined, e.g., job schedules, machine schedules and time schedules. Each one of these schedules puts the set of all operations or operation segments in a given order. For example, a job schedule orders the operations of a given job according to their starting times on the respective machines. A machine schedule refers to a schedule of operations or operation segments on a specific machine. A time schedule may sort all operations of all jobs on all machines according to either their completion times or their starting times.

5. Object base design

An object base in a scheduling system is the module in which the schedule objects are stored and manipulated. One of the main functions of the object base is to store the definitions of all object types, i.e., the object library, and instantiate the objects when needed. Unlike traditional databases which are only capable of storing data, an object base can store the corresponding methods as well. The objects in an object base are stored in a hierarchical structure.

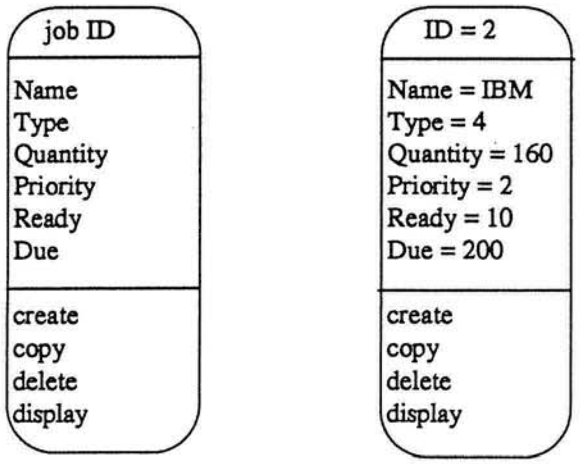
In a conventional relational database, a data type can be defined as a schema of data. For example, a data type "job" can be defined as in figure 3(a) and an instance of the datatype can be as in figure 3(b).

ID	Name	Type	Quantity	Priority	Ready	Due

(a) job data type

2	IBM	4	160	2	10	200

(b) a job instance



(c) job object type and a job object

Figure 3. Job data type and instance.

In the same vein, object types and corresponding instances can be defined. For example, an object type "job" can be defined in the same way as above and $\text{job}_1,\ldots,\text{job}_n$ are instances. All the n instances have the same attributes and the same methods can be applied to all instances.

As stated in the introduction, there are two crucial relationships between object types, namely, the "is-a" relationship and the "has-a" relationship.

An is-a relationship indicates a generalization and the two object types have similar characteristics. Often, the two object types are referred to as a subtype and a supertype. For example, a "machine" object type is a special kind of "resource" object type. A "tool" object type is a different type of machine object type.

A has-a relationship is an aggregation relationship and one object type comprises a number of other object types. A "workcenter" object is composed of several machine objects. A "plant" object comprises a number of workcenter objects. A "routing table" object has job objects as well as machine objects.

Object types related by is-a or has-a relationships have similar characteristics that include attributes as well as methods. In other words, all the attributes and methods of the supertype objects will appear and be used by the corresponding subtypes. For example, a machine object has all the attributes of a resource object and may also have some additional attributes. In a similar way, a workcenter object has all the attributes of a machine object and all the methods applicable to the machine object can be applied to the workcenter object as well. This is often referred to as *inheritance.*

Another concept that is important in object base design is the concept of *template.* A template is an abstract description of a method for different object types. The actual action will be taken according to the object type applied to. For example, the preprocess duplicate method can be defined as follows:

```
construct (object type undecided)
{
  create object1 of object type undecided
}
```

The action performed by this method depends on the object type. A template may not be valid for all the object types and may have some restrictions. There may also be a different interpretation of the template dependent on the object type that it is applied to. For example, splitting a tree object and splitting a set object has different interpretations.

An object can be retrieved and its methods can be triggered through commands which are similar to SQL commands in relational databases. The format of such a command is defined as follows:

Type
Object
Method
Message (*object*, *variable*, *constant*, *etc.*)

A hierarchical structure can be constructed which comprises all object types and relations. In this hierarchical structure, there are two objects that are of significant importance because of their interaction with the schedule generator and the user interfaces (see figure 4).

(i) The *Process Manager* object is an object that controls a number of different process objects. A process object contains all information regarding a machine environment, a set of jobs, processing constraints, and so on. That is, a process object contains all the information corresponding to a particular scheduling problem. However, a process object does not contain the objectives; the objectives are selected by the user (through a particular interface) and the algorithms in the schedule generator are then reconfigured accordingly.

(ii) The *Schedule Manager* object is an object that manages a number of different schedule objects. A schedule object contains all the information regarding a particular (possibly partial) schedule. A schedule can be retrieved, manipulated, and

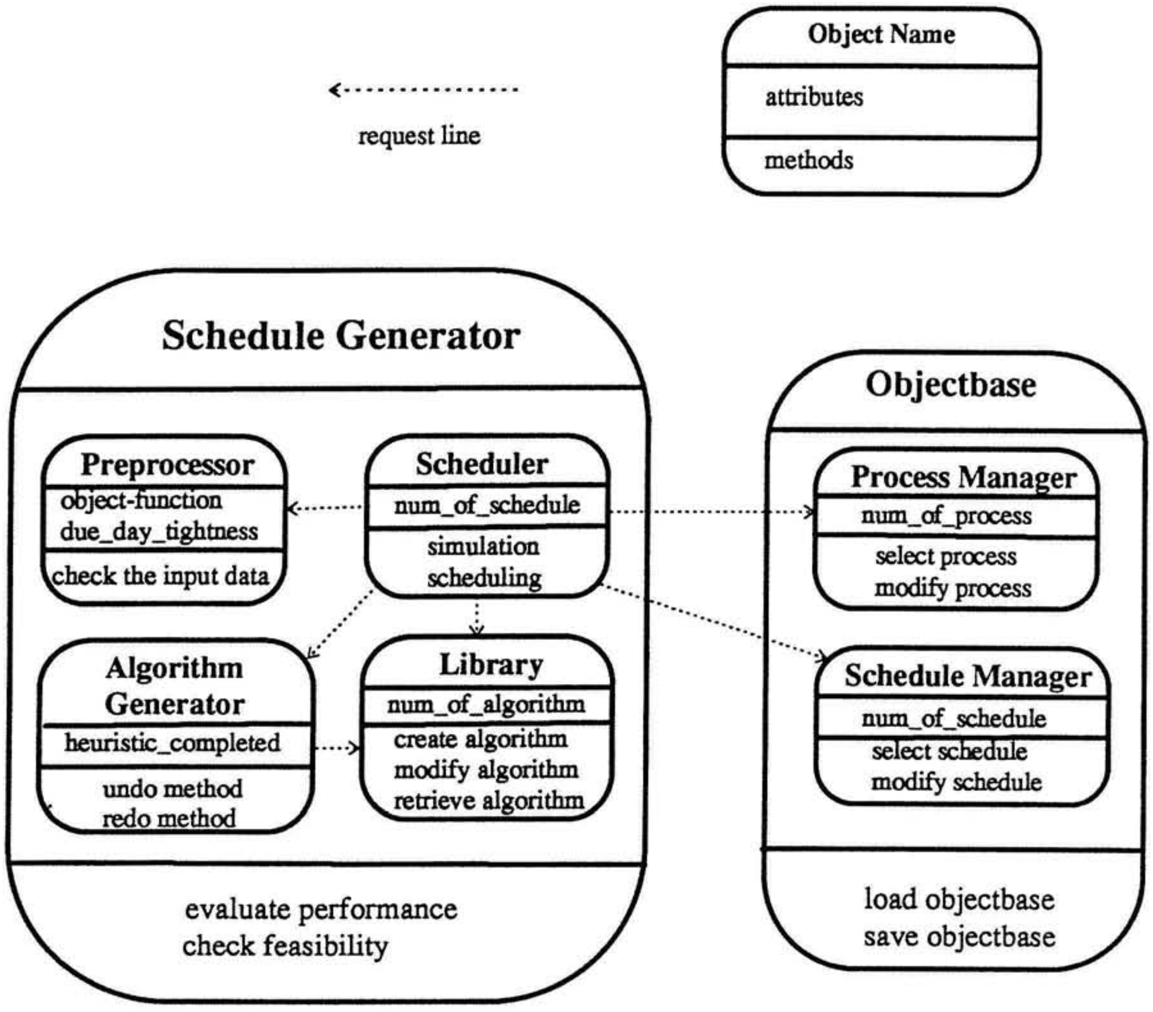


Figure 4. Schedule generator object.

stored. There may be a number of different schedules stored in the schedule manager, since the user may want to compare various schedules or may want to save contingency schedules. With each schedule, there may be a number of different types of associated subschedules, namely job schedules, machine schedules, stage schedules, time schedules, and so on.

6. Schedule manipulation methods

In the previous section, all the basic objects corresponding to a schedule, e.g., machine, operation, and so on, have been defined. In this section, a number of methods are described that are useful for the manipulation of the schedules. The schedules created by a scheduling system are usually not acceptable to the scheduler. The user often wishes to modify the schedules created in order to take preferences or constraints into account which were not incorporated in the system. In order for him to be able to modify the schedule, he must have at his disposal a number of methods that enable him to manipulate the schedule. As these manipulation techniques constitute an important part of the scheduling system, it is of importance to standardize the design of these methods. In this section, some of the more important manipulation methods are described.

Each method can perform a basic manipulation of the schedule. The user, through the user interface, can perform each such manipulation separately and sequentially. In what follows, an overview of the most basic methods is presented. In each scheduling system, there may be additional methods for which the functions depend on the particular system.

The two most basic methods are the *Forward Shift* method (FSM) and the *Backward Shift* method (BSM).

If the FSM is applied to a particular operation on a given machine, then the processing of this operation on the machine is postponed as long as possible. This postponement of the operation may bring about the postponement of subsequent operations on the same machine. The duration of the postponement may be determined by any one of a number of events. For example, the operation on which the FSM is applied or one of the operations following this operation may hit a due date. This due date may be determined by the starting time of the processing of an operation on another (downstream) machine. The FSM applied to a particular operation on a machine will not change any of the processing times of operations on any other machine; it will only affect the processing times on the machine in question. It will not create any (new) infeasibilities or conflicts. The effects of the BSM method are similar to the effects of the FSM method.

The following two methods are concerned with the preemption of operations. The *Split* method splits an operation or operation segment into two smaller operation segments. The *Merge* method takes two operation segments and coalesces them into one.

The next pair of methods are needed in order to change processing sequences. The *Delete* method takes an operation or operation segment from a machine and puts it in a buffer. An operation in the buffer may be depicted in a given way on the screen, but the operation does not appear on any machine. The *Insert* method is used to take an operation or operation segment, either from a buffer or from a machine, and insert it in some time slot on a machine.

A method of a fairly complex design is the *Interchange* method. This method is applied to a pair of operations or operation segments. The positions of the two operation (segments) within a schedule are interchanged. In the design of this method a number of issues have to be dealt with. If the two operation (segments) are of the same size, then the interchange is easy and there are no complications. However, the two operation segments may not be of the same size. Inserting the larger operation segment in the position of the smaller segment may require forward or backward shifting of preceding and succeeding operations. A straightforward rule for the design of the interchange method could be the following: the operation segment that is the first one to start among the two operation segments interchanged is started as early as possible without moving any one of the preceding operations. All operations following this operation segment are started either at their original starting time or as early as possible (i.e., the maximum of two times). The other operation segment involved in the interchange is started as early as possible. So, after performing an interchange a scheduler may apply the FSM and BSM methods on the operations on one or more machines. However, the forward shifting may be constrained by due dates and the backward shifting may be constrained by release dates. (The release dates and the due dates are attributes of the job object type.)

7. Schedule generator design

The schedule generator module is usually the most intricate module of a scheduling system. In operation, the generator has to be extremely flexible since the number of different scheduling situations as well as the variety of available algorithms is immense.

The module consists of a number of objects and methods that have to work in harmony. The entire module may be regarded as an agent object. A more detailed description of the various objects follows.

(i) The *Preprocessor* object. In the preprocessor, the input data of the instance to be scheduled are analyzed. A statistical analysis is done to determine due date tightness, congestion factors, setup time factors, and other statistics of interest that are useful in the generation of the schedules in the other segments of the schedule generator module.

(ii) The *Algorithm Library*. The algorithm library consists of a number of algorithmic procedures. These procedures range from the very simple to the very sophisticated or abstract. A very simple procedure can be a *SORT*, which merely orders a given set of jobs in increasing or decreasing order of a given parameter. A very sophisticated procedure can be a complicated branch and bound algorithm that may only be applicable to a given machine environment. All procedures are basically methods that can be applied to given objects.

(iii) The *Algorithm Generator* object allows the user to construct a new algorithm by combining the methods available in the algorithm library. Also, the user may want to apply the various procedures available in the library interactively. After seeing the results of one procedure, he can decide which procedure to apply next. This object communicates intensively with the user interface objects. For example, the user specifies the objective(s) that have to be minimized; the overall objective is often a weighted average of a number of standard objectives. Through an appropriately designed user interface, the user enters the weights of the various performance measures.

(iv) The *Scheduler and Dispatcher* object is an agent object that generates all sequences and schedules with the specified algorithms. This submodule may work interactively. The user applies the algorithm(s) on a subset of jobs or a subset of machines, contemplates the results and then takes further action. There are a number of methods in a schedule generator. The most important methods are listed below.

(i) The *Simulator* method. After all sequences and schedules have been determined, the simulator submodule determines the entire process plan, i.e., it determines the starting times and completion times of all the operations on all the machines. The simulation of the entire process plan is necessary for a number of reasons: First, it provides all the data needed to compute the performance measures. Second, it is needed in the displays of the user interfaces which enable the scheduler to manipulate the schedules manually.

(ii) The *Performance Measures Evaluation* method. This method receives all the data generated by the simulator and computes the values of all the objective functions of interest.

The interactions and messages between all the objects and methods of the schedule generator are depicted in figure 4.

Of all the objects in the schedule generator, the algorithm generator object is the most complicated one. In the remaining part of this section, this object is described in more detail.

The algorithm generator functions by applying methods to objects. The objects can be any one of the ones described in the previous sections: a set of operations, a set of jobs, a set of machines, and so on. The methods range from the very simple to the very sophisticated. Examples of the very simple methods have already been described in section 3. For example, the sort method can order a set of jobs according to a certain attribute or value. The max method searches for a job with a maximum value of a certain attribute. However, with these basic methods only very simple algorithms can be constructed, e.g., dispatching rules (still, it is possible nowadays to design fairly sophisticated composite dispatching rules that are quite flexible). Schedule generators usually have to be able to perform algorithmic procedures which are significantly more complicated than simple dispatching rules.

Two very common procedures for dealing with scheduling problems are branch and bound and beam search, which is a derivative of branch and bound. One method that is very useful for these procedures is the *branch* method. The branch method is applied to a partial schedule object when a number of jobs or operations still have to be scheduled. The branch method, applied to a partial schedule, generates a number of extended partial schedules. Each generated partial schedule is obtained by adding one job (or operation) to the original partial schedule. Branch and bound approaches as well as beam search approaches can be constructed easily with such a branch method.

Many scheduling algorithms are based on decomposition techniques (e.g., a dynamic programming is basically an algorithm based on decomposition techniques). In order to construct algorithms of this class, a *decompose* method is very useful. The decompose method splits an existing set of jobs or an existing set of machines into a number of subsets. Each subset is then scheduled separately. Either a subset of the jobs or a partial schedule is then an object. The object, in addition to the set of jobs, must contain then also information with regard to the earliest possible time that each machine is available for the jobs in the subset.

Another important method, which is basically the counterpart of the decompose method, is the *compose* method. After partial schedules have been found for the various subsets, these partial schedules have to be coalesced into a single integrated schedule. The compose method applied to a number of objects (each one of these being a partial schedule) generates an integrated schedule.

The algorithm generator allows a sophisticated scheduler (or his technical support) to construct a new algorithmic framework by combining the various methods at his disposal. For example, the scheduler may want to combine a certain preprocessor object with a beam search technique followed by a neighbourhood search technique.

8. User interface design

The traditional user interface used to be a fairly simple textual or numerical display of input and output. The types of information that could be expressed this way are usually very limited. Modern user interface techniques are based on hypertext techniques where the display can be shown in text graphics, images, and combinations of these (see, for example, Woerner and Biefeld [13] for such interfaces in scheduling systems).

The following classification can be made of some of the more basic user interface objects.

(1) *Message*: a message displays a single line of alphanumeric information. A message may be in the form of a question requiring a yes/no answer.

(2) *Multiple-line message*: a multiple-line message is conceptually similar to a single-line message, except that multiple lines of information are allowed.

(3) *Choice*: a choice can be made by means of a button, a checkbox or a toggle. A choice can be made exclusively or non-exclusively.

(4) *Menu*: a menu is typically used for selection of functions and not for the display of information. The functions can be nested.

(5) *Dialogue Box*: a dialogue box is a generalization of a multiple-line message. It usually includes a scroll bar and supports a select function. Normally, it is used for the display of short items.

(6) *Text Window*: a text window is similar to a dialogue box, except that it supports longer message displays and file handling functions.

(7) *Canvas*: a canvas can be used for graphical displays.

(8) *Frame*: a frame can play the role of the base window for all other kinds of interface objects.

Through combinations of these primitive objects, more complicated interface objects can be constructed. Specific methods can be defined for the same object for different display purposes. For example, a canvas object can be used for the display of either a bar chart or a pie chart.

The interfaces which display schedule information may take many different forms. All these interface objects can be constructed using the primitive objects defined in the beginning of this section. Some of the basic interfaces displaying schedule information are:

(1) *The Order and Job Information interface.* This interface consists of multiple messages regarding the attributes of the order objects and the job objects. This interface often uses menus, dialogue boxes and text windows.

(2) *The Gantt Chart interface.* The Gantt chart interface is a canvas with methods for drawing Gantt charts, with function buttons, and with messages regarding the status of the schedule.

(3) *The Machine Dispatch List interface.* This interface has to be a multiple text window (it has to display for each machine a list of the jobs in the order in which they are to be processed), with function buttons and with messages for the display of the schedule status.

(4) *The Capacity Buckets interface.* The construction of this interface is similar to the construction of the Gantt chart interface.

(5) *The Throughput Diagram interface.* This interface depicts, usually in a graphical way, the number of jobs that have not started with their processing yet, the number of jobs currently in the system and the number of jobs which have already completed their processing.

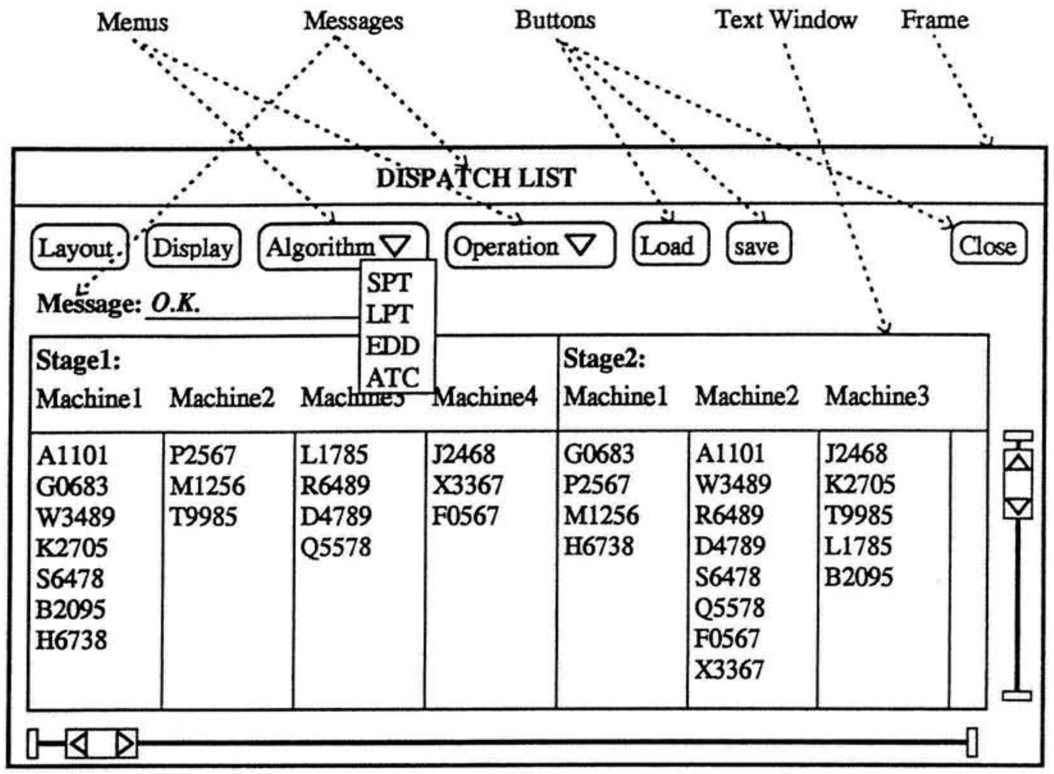
(6) *The Performance Evaluation interface.* This interface displays the values of all the performance measures of interest. These measures may be displayed either in alphanumeric characters or in the form of bars.

In figure 5, the dispatch list interface object is depicted. This object is constructed using menus, buttons, text windows, messages, and so on.

All interfaces described above are used for the display of data regarding the schedule, often referred to as dynamic data. The static data, i.e., order, job and machine information, do not depend on the schedule; they may often be displayed through appropriate interfaces as well. Examples of such interfaces are:

(1) *The Plant Layout interface.* This interface may depict graphically the workcenters and the machines in the plant as well as the possible routes between the workcenters.

(2) *The Resource Calendar interface.* The resource calendar displays shift schedules, holidays, preventive maintenance schedules of the machines, and so on.



DISPATCH LIST

Message: ***O.K.***

Stage1:				**Stage2:**		
Machine1	Machine2	Machine3	Machine4	Machine1	Machine2	Machine3
A1101	P2567	L1785	J2468	G0683	A1101	J2468
G0683	M1256	R6489	X3367	P2567	W3489	K2705
W3489	T9985	D4789	F0567	M1256	R6489	T9985
K2705		Q5578		H6738	D4789	L1785
S6478					S6478	B2095
B2095					Q5578	
H6738					F0567	
					X3367	

Figure 5. Dispatch list user interface object.

(3) *The Routing Table interface.* The routing table depicts static data (which are independent of the schedule) that are heavily job and machine dependent.

Besides the interfaces described above which are needed in most scheduling environments, there are often interfaces that are heavily application dependent. Schedule interfaces can also be classified according to the semantic meaning of information.

Interface objects are closely associated with other schedule objects. The links between interface objects and other objects (e.g., in the object base) are established by messages which pass from one object to another. There are two types of methods that control these messages, namely response methods and retrieval methods.

The response methods, which in user interface jargon are often called *callback* functions, trigger the interface objects. For example, pushing a button or selecting an item from a dialogue box may activate such a method.

The retrieval methods are concerned with the update of data in the interface objects. Sometimes, these methods are embedded in the response methods.

The display methods can pass messages (data) to the interface objects in order to activate corresponding functions. The triggered interface objects may also send messages to schedule objects and may receive messages from schedule objects (see figure 6).

Usually, each schedule object has its own interface object to take care of the display of the input/output information or to activate other methods. Sometimes, more

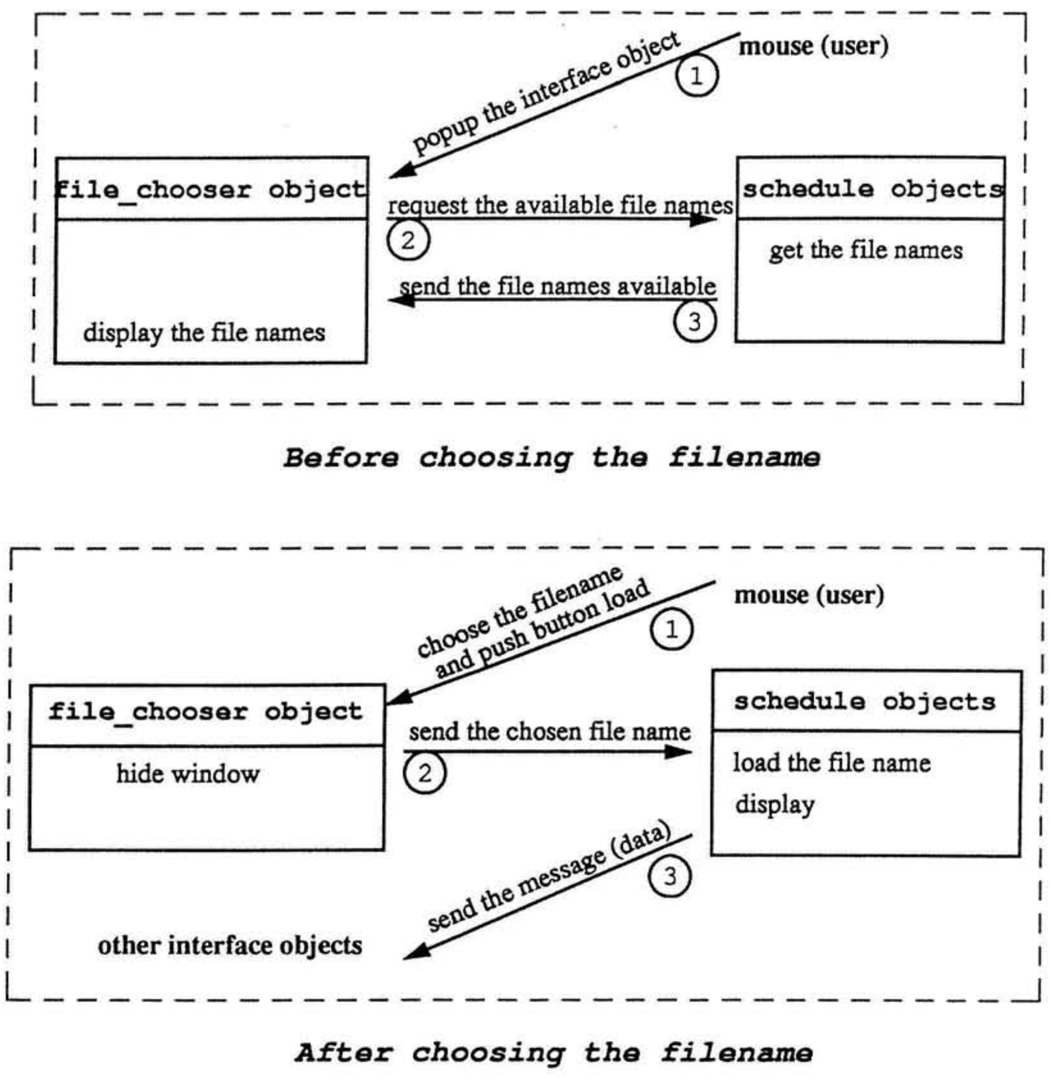


Figure 6. Interaction between interface object and schedule object.

than one interface object is associated with a schedule object, or vice versa. There are some common rules for linking interface objects with methods for corresponding schedule objects, namely,

(i) message object: when the input parameter of a method is a number or a string, e.g., the ready times or release dates of jobs;

(ii) dialogue object: when the input parameter or the output of a method can be listed, e.g., the list in which the jobs are dispatched;

(iii) button object: when a method does not have an input parameter, e.g., when it has to be decided whether or not a SORT routine has to be applied;

(iv) menu object: when a method does not have an input parameter, but has different modes of functionality, e.g., certain scheduling algorithms can be applied in different ways through the use of different bounding techniques, and so on;

(v) canvas object: when the output of a method can be displayed in the form of graphs or bars, e.g., a Gantt chart;

(vi) textwindow: when the input and the output of a method are multiple fields and multiple lines, e.g., a table of starting times and completion times of the jobs.

For the composite interface objects, we can have more variations to create more sophisticated links between interface objects and schedule objects. Usually, we use a single frame object as a base for a composite interface object and link these with the corresponding schedule objects.

We may combine interface objects with corresponding schedule objects to form more general objects. For example, a layout object may have a corresponding interface object for the display of layout information and handle activation of its methods, and we can put these together to construct a more general layout object with all the interface functions.

9. Discussion

In this paper, we described the architecture of an object-oriented scheduling system. The goal of the design was to define objects for *every* aspect of the scheduling system. As such, it promotes an extremely modular design which makes the system easily reconfigurable and extendable. This approach is not necessarily superior to other approaches; the design considerations presented here can be used in conjunction with other approaches.

A system has been built according to these guidelines. The system, CUISE 3.0, is in operation at Columbia University and several of its modules have been implemented at an industrial site. CUISE 3.0 is an extension of the system CUISE 2.0 described by Pinedo [7].

A follow up in this direction of research is the development of a higher-level language specifically designed for the development of scheduling systems. The advantages of such a language are clear. It would significantly shorten the development time and prototyping time. It also would enable easier extensions and modifications of the system by people with less knowledge of the inner details of the system.

Acknowledgement

We are grateful to one of the referees for his careful scrutiny and his many helpful remarks which improved the presentation considerably.

References

[1] G. Booch, *Object-Oriented Analysis and Design with Application*, 2nd ed., Benjamin Cummings, 1994.

[2] A. Collinot, C. LePape and G. Pinoteau, SONIA: A knowledge-based scheduling system, Artificial Intelligence in Engineering 2(1988)86–94.

[3] M.S. Fox and S.F. Smith, ISIS – A knowledge-based system for factory scheduling, Expert Systems 1(1984)25–49.

[4] P. LeFrancois, M.H. Jobin and B. Montreuil, An object-oriented knowledge representation in real time scheduling, in: *New Directions for Operations Research in Manufacturing*, G. Fandel, Th. Gulledge and A. Jones, eds., Springer, New York, 1991, pp. 262–279.

[5] J. Martin, *Principles of Object-Oriented Analysis and Design*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

[6] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[7] M. Pinedo, *Scheduling: Theory, Algorithms and Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1995.

[8] M. Pinedo, R. Samroengraja and B.P.-C. Yen, Design issues with regard to scheduling in manufacturing, in: *Control and Dynamic Systems Vol. 60*, C. Leondes, ed., Academic Press, San Diego, 1993, pp. 203–238.

[9] J. Sauer, Dynamic scheduling knowledge for meta-scheduling, *Proceedings of the 6th International Conference on Industrial Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)*, Edinburgh, Scotland, 1993.

[10] S.F. Smith, Knowledge-based production management: Approaches, results and prospects, Production Planning and Control 3(1992)350–380.

[11] S.F. Smith, N. Muscettola, D.C. Matthys, P.S. Ow and Y. Potvin, OPIS: An opportunistic factory scheduling system, in: *Proceedings of the 3rd International Conference on Industrial and Expert Systems (IEA/AIE 90)*, Charleston, SC, 1990.

[12] S.F. Smith and O. Lassila, Configurable systems for reactive production management, in: *Knowledge-Based Reactive Scheduling (B-15)*, E. Szelke and R.M. Kerr, eds., Elsevier Science, North-Holland, 1994, pp. 93–106.

[13] I.W. Woerner and E. Biefeld, Hypertext-based design of a user interface for scheduling, in: *Proceedings of the AIAA Computing in Aerospace 9*, San Diego, CA, 1994.

[14] B.P.-C. Yen, On the architecture of object-oriented scheduling systems, Ph.D Thesis, Department of Industrial Engineering and Operations Research, Columbia University, New York, NY, 1995.

[15] E. Yourdon, *Object-Oriented System Design: An Integrated Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1994.