ABSTRACT

A FRAMEWORK FOR SCHEDULING ALGORITHMS IN MANUFACTURING ENVIRONMENT

By

Eduardo de Aquino Martorano

December 2010

The purpose of this work is to design and implement a domain specific framework to be used in the implementation of scheduling algorithms for manufacturing environments. Developers of scheduling algorithms are faced with several tasks that are inherent to any software development, but do not add value to the problem being addressed. Moreover, different scheduling algorithms present some similarities, for example, they might share a uniform data model. In light of these considerations, a framework has been created to provide functionalities commonly encountered in the implementation of scheduling algorithms. This framework has been tested in the implementation of a number of scheduling algorithms, showing that its use greatly simplifies the implementation task, allowing developers to focus on the algorithm itself instead of other peripheral tasks.

A FRAMEWORK FOR SCHEDULING ALGORITHMS IN MANUFACTURING ENVIRONMENT

A THESIS

Presented to the Department of Computer Engineering and Computer Science

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

Committee Members:

Shui Lam, Ph. D. (Chair)
Burkhard Englert, Ph. D.
Xiaolong Wu, Ph. D.

College Designee:

Ken James, Ph. D.

By Eduardo de Aquino Martorano

B.E., 1998, Pontifícia Universidade Católica de São Paulo, Brazil

December 2010

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Shui Lam, for all her help.

Very special thanks to all my family, my lovely parents, and specially my brother, Marcelo Martorano, who has guided me so much during all my student life.

To my wife Silvia.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Scheduling of jobs in the manufacturing environment is a crucial task to be executed during production planning. The execution of any production process cannot start before a feasible schedule has been created.

It is common for organizations to create orders (jobs) which are going to be executed in near future in their facilities. These orders are usually created based on customer requests (make-to-order) or the need to balance stock levels (make-to-stock). The creation of orders, however, does not specify in which sequence the new orders will be executed or how the company's resources will be allocated to execute them.

Once a certain number of orders have been created, the next step is to schedule them. Many definitions of scheduling can be found in the literature; an illustrative example is given below (1):

"Scheduling is to forecast the processing of a work by assigning resources to tasks and fixing their start times [...] The different components of a scheduling problem are the tasks, the potential constraints, the resources and the objective function [...] The task must be programmed to optimize a specific objective [...] Of course, often it will be more realistic in practice to consider several criteria."

This definition can be easily applied to any manufacturing environment. Tasks are usually orders or operations of an order. Complex orders are usually broken down into several operations and each of them is a job or task to be scheduled. Very frequently those orders and operations have specific requirements. A good requirement example is the sequence of operations for an order. There are certain manufacturing environments, such as flow shops, where jobs have a pre-defined sequence of operations. This sequence is a constraint that must be satisfied by the schedule. A problem is considered not feasible if no solution can be found without violating the constraints.

Examples of resources are tools, machines, people, and work centers. They have limited availability and cannot be shared. They can be allocated to a single job at a certain point in time. Jobs need resources in order to be executed. The process of manufacturing a finished product might require the use of a welding machine for half hour, a drilling machine for one hour and so on.

Finally, organizations want to operate their production systems to minimize their costs, which usually means minimizing makespan[1] and late orders (lateness), maximizing resource utilization, and other types of optimization. The cost function to be optimized is called the objective function. Often, scheduling problems involve not one but multiple objective functions. Industries try to minimize their makespan, their late others, and their amount of WIP (work-in-process) inventory, all at the same time. This complete minimization is not always possible and, therefore, a trade-off process is applied. The

[1] Makespan is the time difference between the start and finish of a sequence of jobs or tasks.

challenge of any scheduling algorithm is to best allocate the available resources in order to minimize as many objective functions as possible without violating any existing constraints.

The study of scheduling algorithms has been very prolific and generated an innumerous number of publications in the last 50 years. The number and variety of scheduling models in the literature is astounding (2). Several different models have been developed for problems in the field of deterministic scheduling as well as stochastic scheduling. Algorithms such as Lawler's algorithm or Johnson's algorithm are classical examples. Although scheduling algorithms might be very different, because they frequently address different problems, there are certain similarities between them. Those similarities are not only present in the way the jobs, machines, and constraints are represented, but also in certain blocks of logic, which sometimes are re-used by different algorithms.

The many similarities found among scheduling problems have led researchers in the direction of creating a unified data model capable of accommodating a large set of different problems. There are ontologies[2] for scheduling problems, such as the one proposed by Rajpathak, Motta, and Roy (3), or that created by Smith and Becker (4). Those are examples of the work that has been done in order to create a data model that could be used throughout many different domains. Their work has focused on creating a comprehensive task-centric model to support scheduling problems regardless of their

[2] Ontology can be seen as a reference model to describe the entities which exist in a universe of discourse and their properties (9).

domain. They have proved that, despite of the business domain, scheduling problems share common properties; therefore, a common model can be used. The foremost objective of creating a comprehensive data model is reusability. Once a data model has been created, developers can use it while building new applications, saving a significant amount of time.

Although ontologies, such as the ones mentioned earlier, can significantly help the implementation of different applications, they are pure abstract models and no real implementation has been provided. The decision of providing only abstract entities was done intentionally in order to create an independent model that can be used across different platforms. That leaves developers with the task of creating their concrete implementations. Moreover, the researches mentioned previously focused only on the scheduling domain and no attention was given to other artifacts that could be of great value to developers.

A framework should support not only operations present in the business domain for which it has been designed, but also other basic operations that are inherent to the process of application development. Most applications have to deal with certain number of operations, such as saving data to a database repository, log information for debug purpose, and data validations. These operations can and should be implemented at the framework level and should be reused by many different projects.

The intent of this work is to model and construct a basic framework to be used in the implementation of scheduling algorithms. It will be shown that the creation of a domain specific framework for scheduling applications is a feasible task. Some concrete examples will be provided to support this assumption.

CHAPTER 2

OBJECTIVE

Developers of computer applications face, quite often, an enormous task when implementing any new software to address a given problem. Their workload is not only generated by the problem which they are trying to address, but also but also by a number of other implicit tasks required to create a useful computer application.

User interface, data validation, storage support are a few of the non-value added tasks among many others which can be found on almost every software development. The "non-value added" adjective has been used, because those tasks do not add any real value to the solution of the problem itself. They are there because, without them, a usable application would not exist.

Those tasks, although not direct related to the main problem being addressed, are responsible for a very significant amount of time and money spent to build any application. This is so frequent that developers have tried to minimize the overhead of those non-value added tasks by using component libraries. There are user interface component libraries for helping users to create data entry applications. There are mathematical libraries with an innumerous number of mathematical functions. Those two are only one example of libraries available today.

Scheduling applications for manufacturing, like any other computer application, suffer the effect of those non-value added tasks. In addition, a simple analysis of scheduling applications for distinct algorithms shows a very important aspect: their non-value added tasks share many similarities.

Regardless of the type of shop floor configuration in the manufacturing facility, the objective of any scheduling application is to best allocate the facility's resources, such as machines and people, to their production. Distinct scheduling algorithms must provide the users with an interface to configure the environment or to enter orders to be scheduled. That is just one simple example where the overlap between those non-value added tasks can be found.

The cost and time of implementing scheduling algorithms could be substantially minimized if developers had available a domain specific framework for scheduling applications. Domain specific frameworks are a set of software components developed for a specific business domain. This framework could provide out-of-the-box[3] support for many tasks that are required to implement scheduling applications.

The objective of the present work is to propose, develop, and test a general framework to support the implementation of scheduling algorithms for manufacturing environments. This framework provides important features, such as, data structures, dispatching rules, common scheduling algorithms operations, exception handling, user interface (UI) capabilities for entering data and displaying results and extensibility. Its

[3] "Out of the box" is also used as a synonym for "off the shelf," meaning a ready-made software, hardware, or combination package that meets a need that would otherwise require a special development effort.

infra-structure enables developers to completely focus on implementing the scheduling algorithm instead of any other peripheral tasks inherent to application development.

CHAPTER 3

BACKGROUND

Notation

The vast number of scheduling problems has led researches to create a specific notation that can be used in the classification of scheduling problems. Graham has introduced a 3-field notation that is widely used in the literature (5).

Jobs and machines are the foremost entities present in Graham's notation. Jobs are often called tasks or activities. It is assumed that a job has a certain processing requirement and can be processed on at most one machine at a time. In manufacturing lingo, order or operation are often the preferred replacements for job although this choice might generate certain confusion when describing problems where jobs have several operations such as in flow shop problems. Resource is used when referring to machines due to the fact that machines are a specific type of resource. People, tools, work centers are other types of resources. Resources are finite and the utmost objective of any scheduling algorithm is to best allocate them.

The 3-field notation is composed of α, β, and γ ($\alpha | \beta | \gamma$): α describes the machine environment, β job characteristics and γ the criterion or criteria to be optimized (6).

The machine environment commonly defined by $\alpha = \alpha_1 \alpha_2$. Table 1 presents the most common values for $\alpha_1 \alpha_2$.

TABLE 1. Machine Environment

α	Environment
$\alpha_1 = \emptyset$	Single Machine
$\alpha_1 = P$	Identical Parallel Machines
$\alpha_1 = Q$	Uniform Parallel Machines
$\alpha_1 = R$	Unrelated Parallel Machines
$\alpha_1 = O$	Open shop
$\alpha_1 = F$	Flow shop
$\alpha_1 = J$	Job shop
$\alpha_2 = m$	m is the number of machines

Job Characteristics (β): $\beta = \beta_1\ \beta_2\ \beta_3\ \beta_4\ \beta_5\ \beta_6$

β_1 = pmtn (preemption) Job preemption is allowed

β_2 = res: when present, it indicates that there are resource constraints such as non renewable resources

β_3 = {prec, tree}: Type of precedence

β_4 = when $\emptyset$, jobs have same release date

β_5 = when $\emptyset$, jobs have arbitrary processing times

β_6 = when $\emptyset$, jobs don't have deadlines

Optimal criteria (γ): $\gamma \varepsilon \{f_{\max}, \Sigma f_j\}$. The optimal criteria usually involves minimizing a max functions ($f_{\max}$) or minimizing a sum function (Σf_j).

A common $f_{\max}$ is $C_{\max}$=max (C_i) where C_i represents the completion time of job J_i. Several other objective functions such as $L_{\max}$ (maximum lateness), ΣC_i (total completion time) or $T_{\max}$ (maximum tardiness) are also commonly found.

Objective Functions

Objective functions are the main artifact available to measure the outcome generated by any scheduling algorithm. There are many well known objectives such as "Total Completion Time" or "Number of Late Jobs" that are often used. However, the number of objectives is virtually infinite and often varies from company to company as pointed by Oyetunji (7). Table 2 presents the most common objective functions.

Scheduling Typology

A typology is a classification of the problems according to their nature (1). Figure 1 shows a general typology. The majority of the scheduling problems can be placed in one of the categories presented in Figure 1. The type of a problem is an important aspect to take in consideration while analyzing the problem classification. The categories listed above have been broken down in 3 different types: scheduling, scheduling and assignment with stages and general scheduling and assignment (1).

The objective of scheduling problems without assignment is usually to determine the start date for every job. Every operation can be processed by only one machine therefore assignment is not to be determined by the scheduling algorithm. In the second category, scheduling and assignment with stage, a certain number of machines are available for each operation (stage). General scheduling and assignment are a superset of the former where machines are not restricted to only one stage.

TABLE 2. Common Objective Functions

Objective	Formula	Description
Total completion time	$\sum_{i=1}^{n} Ci$	Sum of completion times.
Total weighed completion time	$\sum_{i=1}^{n} WiCi$	Sum of completion times multiplied by their job weights.
Average completion time	$\frac{1}{n}\sum_{i=1}^{n} Ci$	Sum of completion times divided by the number of jobs.
Average weighed completion time	$\frac{1}{n}\sum_{i=1}^{n} WiCi$	Sum of completion times multiplied by job weights divided by the number of jobs.
Maximum completion time	$\mathrm{Max}(C_1, C_2, \ldots, C_n)$	Completion time of the last job. Also called makespan.
Total flow time	$\sum_{i=1}^{n} (Ci - ri)$	Sum of completion times minus release times.
Total weighed flow time	$\sum_{i=1}^{n} Wi(Ci - ri)$	Sum of completion times minus release times multiplied by job weights.
Average flow time	$\frac{1}{n}\sum_{i=1}^{n} (Ci - ri)$	Total flow time divided by the number of jobs.
Average weighed flow time	$\frac{1}{n}\sum_{i=1}^{n} Wi(Ci - ri)$	Total weighed flow time divided by the number of jobs.
Total lateness	$\sum_{i=1}^{n} (Ci - di)$	Sum of completion times minus job due date.
Total weighed lateness	$\sum_{i=1}^{n} Wi(Ci - di)$	Sum of completion times minus due date multiplied by job weights.
Average lateness	$\frac{1}{n}\sum_{i=1}^{n} (Ci - di)$	Total lateness divided by the number of jobs.
Average weighed lateness	$\frac{1}{n}\sum_{i=1}^{n} Wi(Ci - di)$	Total average lateness divided by the number of jobs.

TABLE 2. Continued

Objective	Formula	Description
Maximum lateness	Max{$(C_1$-$d_1)$,$(C_2$-$d_2)$,…,$(C_n$-$d_n)$}	Biggest lateness among all jobs.
Number of tardy jobs	Let $U(i) = \begin{cases} 1; & Ci > di \\ 0; & otherwise \end{cases}$ $\sum_{i=1}^{n} U(i)$	Number of jobs that have been completed after its due date.
Average number of tardy jobs	$\frac{1}{n}\sum_{i=1}^{n} U(i)$	Number of tardy jobs divided by the number of jobs.
Total tardiness	$\sum_{i=1}^{n} \max\{0, (Ci - di)\}$	Similar to total lateness but it carries only positive values.
Total weighed tardiness	$\sum_{i=1}^{n} \text{Wi}\ [\max\{0, (Ci - di)\}]$	Sum of job tardiness multiplied by their weights.
Average tardiness	$\frac{1}{n}\sum_{i=1}^{n} \max\{0, (Ci - di)\}$	Total tardiness divided by the number of jobs.
Average weighed tardiness	$\frac{1}{n}\sum_{i=1}^{n} \text{Wi}\ [\max\{0, (Ci - di)\}]$	Total weighed tardiness divided by the number of jobs.
Total earliness	$\sum_{i=1}^{n} (di - Ci)$	Sum of job due dates minus completion time.
Total weighed earliness	$\sum_{i=1}^{n} Wi(di - Ci)$	Sum of due date minus completion time multiplied by job weight.
Average earliness	$\frac{1}{n}\sum_{i=1}^{n} (di - Ci)$	Total earliness divided by the number of jobs.
Average weighed earliness	$\frac{1}{n}\sum_{i=1}^{n} Wi(di - Ci)$	Total average earliness divided by the number of jobs.
Maximum earliness	Max{$(d_1$-$C_1)$,$(d_2$-$C_2)$,…,$(d_n$-$C_n)$}	Biggest earliness among all jobs.

TABLE 2. Continued

Objective	Formula	Description
Number of early jobs	Let $U(i) = \begin{cases} 1; & Ci < di \\ 0; & otherwise \end{cases}$ $\sum_{i=1}^{n} U(i)$	Number of jobs that have been completed before their due date.
Average number of early jobs	$\frac{1}{n}\sum_{i=1}^{n} U(i)$	Number of early jobs divided by the number of jobs.

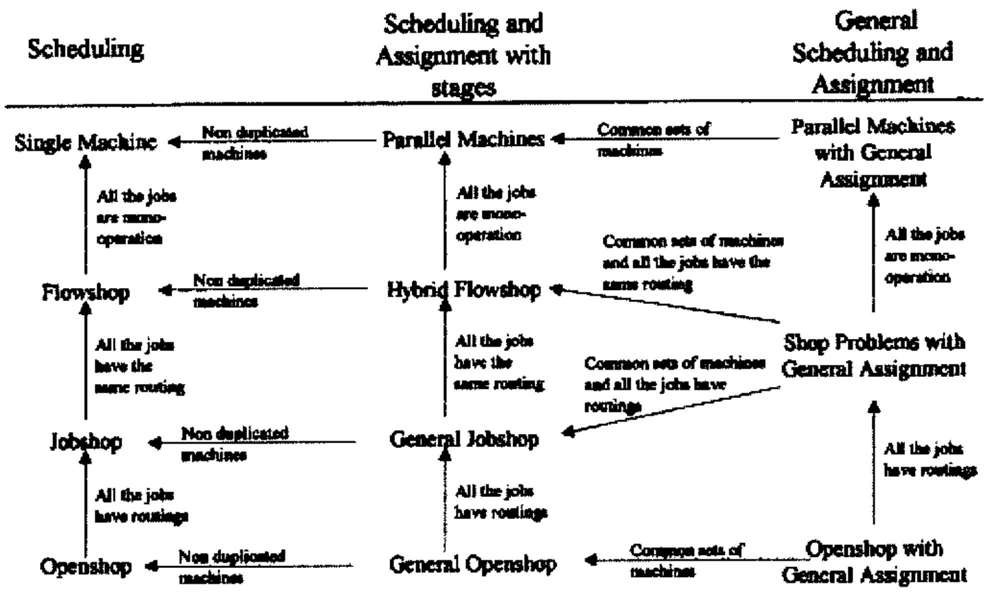

Scheduling
Scheduling and Assignment with stages
General Scheduling and Assignment
Single Machine
Non duplicated machines
Parallel Machines
Common sets of machines
Parallel Machines with General Assignment
All the jobs are mono-operation
All the jobs are mono-operation
Common sets of machines and all the jobs have the same routing
All the jobs are mono-operation
Flowshop
Non duplicated machines
Hybrid Flowshop
All the jobs have the same routing
All the jobs have the same routing
Common sets of machines and all the jobs have routings
Shop Problems with General Assignment
Jobshop
Non duplicated machines
General Jobshop
All the jobs have routings
All the jobs have routings
All the jobs have routings
Openshop
Non duplicated machines
General Openshop
Common sets of machines
Openshop with General Assignment

FIGURE 1. Typology.

Single Machine

Single Machine scheduling problems are those were a single resource is available to execute all the jobs to be scheduled. Jobs consist of a single operation.

Flow Shop

Flow Shop problems are very common in manufacturing environments. Orders (jobs) are defined by a set of operations and each operation must be performed by a different work center. Work centers have just one machine therefore assignment is not required. The execution sequence of the operations is pre-defined and must be followed. Flow shop problems are a specific case of job shop problems where all jobs have the same operation sequence.

Job Shop

Job Shop environments provide the most flexible form of manufacturing (8). A certain number of work centers are available to process orders (jobs). Work centers have a single machine. Jobs consist of a number of ordered operations but unlike flow shop problems, the operation sequence for all jobs is not the same.

Parallel Machine

More than one resource (machine) is available at a given work center in parallel machine environments. Parallel machine problems are usually divided into 3 categories: identical, uniform and unrelated. For identical machines, the processing time of a job is the same on any machine. In uniform problems, machines have different speeds and the processing time of a job on a given machine depends on the machine speed. Unrelated problems are a generalization of the uniform problem. Jobs have different processing time on every machine and that is not dependent on the machine speed. Parallel machine problems share some similarities with single machine problems due to the fact that the jobs have a single operation.

Flexible Flow Shop

Flexible flow shop problems differ from flow shop problems in the work center configuration. While in flow shop problems there is a single machine per work center, in flexible flow shops, there is more than one machine available per work center and operations can be assigned to any machine in the work center.

Flexible Job Shop

The same analogy used between flexible flow shop and flow job can also be applied between flexible job shop and job shop environments. Flexible job shop is, in some ways, similar to job shops regarding job configuration (jobs have different routings) but more than one machine could be available at a given work center.

Dispatching Rules

Pinedo (2) has pointed out that dispatching rules are general purpose procedures that are useful in dealing with scheduling problems in practice and can be implemented with relative ease in industrial scheduling system. Dispatching rules are heuristic solutions that do not guarantee to optimize any objective function, however rules such as the SPT (Shortest Processing Time First rule) when applied to the problem $1||\Sigma C_i$ or EDD (Earliest Due Date First rule) applied to $1|r_i|L_{max}$ end up yielding the optimized schedule. Moreover, dispatching rules are often seeing as part of other scheduling algorithms. Table 3 shows a list of a few dispatching rules.

TABLE 3. Dispatching Rules

Rules	Description
FAM-SPT	First Available Machine Shortest Processing Time First
FAM-LPT	First Available Machine Longest Processing Time First
FAM-WSPT	First Available Machine Weighed Shortest Processing Time First
FAM-EDD	First Available Machine Earliest Due Date First
FAM-EDD	First Available Machine Earliest Due Date

Ontologies

The number of scheduling problems is virtually infinite. Small differences between two very similar scheduling problems may result in a completely new problem with a totally different solution. This intrinsic characteristic of scheduling problems represents a challenge for information systems and makes it hard to build generic solution that can be used, as-is, to solve a large spectrum of problems. However, one might notice that even though problems and solutions are very distinct, a large number of similarities can be found if an analysis is done at entity level for each problem. Entities such as work order, task, job, operation, resource and etc can be found in almost every problem. That leads in the direction of creating a common abstract model to support the implementation of different problems.

Ontologies are abstract model used to represent objects found in a given business domain. There has been several works in the direction of creating ontologies for scheduling problems. The main objective of these works is always to create a very comprehensive and generic model that could be applied to a large set of different scheduling problems. Moreover, these ontologies can work as a bridge between the domain analysis and the system development (4).

Figure 2 shows a task centric ontology created by Rajpathak et al. (9). This ontology could be used by a developer to build a class diagram during the development of his application.

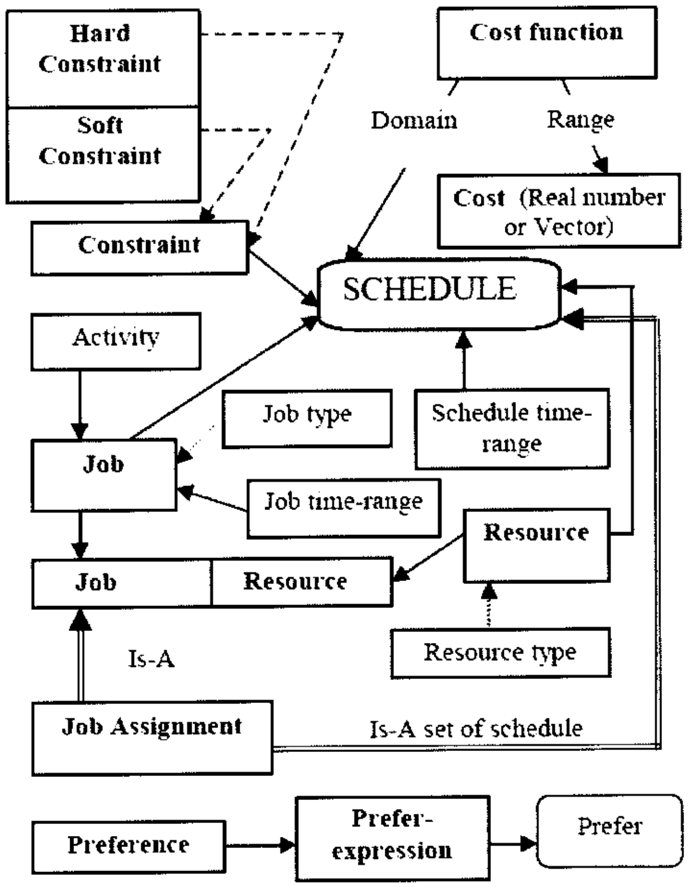


FIGURE 2. Ontology for scheduling application.

Framework Design

Roberts and Johnson (10) have defined a common path for creating frameworks. In their work, frameworks should be built on generalizations created from concrete example, as can be seen from the excerpt given below (10) :

> The ability to generalize for many applications can only come by actually building the applications and determining which abstractions are being reused across the applications. Generalizing from a single application rarely happens. It is much easier to generalize from two applications, but it is still difficult. The general rule is: build an application, build a second application that is slightly different from the first, and finally build a third application that is even more different than the first two. Provided that all of the applications fall within the problem domain, common abstractions will become apparent.

This idea suggests that the best way to develop a framework should follow a cycle where the more concrete implementations have been created; the easier it is for the developer of the framework to see commonalities among these classes. These concrete classes provide ground for generating abstractions and that any attempt to create abstractions without generating concrete implementations would not succeed.

A simple development cycle of a framework could be:

Cycle 1:

Concrete Class 1

Concrete Class 2

Cycle 2:

Base or Abstract class A

Concrete Class 1

Concrete Class 2

Cycle 3:

Base or Abstract class A

Concrete Class 1

Concrete Class 2

Concrete Class 3

Cycle 4:

Re-design Class "A" taking in consideration Class 3.

One important aspect of designing frameworks is in the selection criteria used to select the candidates for the concrete implementation that are going to be used to find the right abstractions. A diverse and heterogeneous selection of problems should be used. The success of the framework is directly linked to how well it can handle different types of the problems. Therefore, the richer the sample selected, the bigger the chances that the framework will support a wide range of different problems without the need of extensions. However, extension cannot ever be avoided due to the fact that is not possible to create classes which could support any problem; therefore the framework should be designed with that in mind.

CHAPTER 4

METHODOLOGY

The process of creating a domain specific framework such as a scheduling framework differs from a generic framework. Generic frameworks like Microsoft . Net and Java do not try to provide a more tailored support to specific problems that are found in certain business domain. Instead, they focus on a common ground type of support that spans throughout many different business domains. Database support, data conversation, network communication, collection manipulation are examples of base functionality provided by non domain oriented frameworks.

Domain specific frameworks can be envisioned as a layer between a generic framework and a business application (Figure 3).

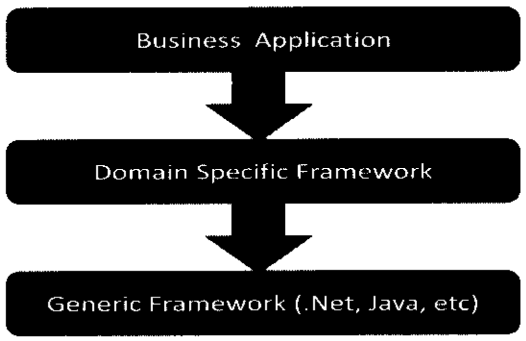

Business Application
Domain Specific Framework
Generic Framework (.Net, Java, etc)

FIGURE 3. Domain specific framework.

A framework for scheduling applications should provide certain functionalities commonly found in many different algorithms. A good example of a common operation found in several distinct algorithms is the sorting operation. Several scheduling algorithms and dispatching rules have some kind of sorting as part of their logic. Therefore, a sorting operation provided at framework level would benefit several different implementations by avoiding the same logic to be re-written many times.

As pointed out by Roberts and Johnson (10), the development of a framework is an expensive project that demands a significant effort. The scope of this project is to implement a certain number of basic functionalities in order to verify how much of the development of scheduling algorithms can be simplified by using a framework that was developed for supporting scheduling applications.

The first step to develop the scheduling framework was to identify the major blocks of functionality that, if provided by the framework, would speed up the implementation of algorithms. In order to identify those blocks, a simple schematic design has been done for the development of two scheduling applications. The assumption during the creating of both designs was that every application would have to be developed entirely from scratch. Lawler's and Moore's algorithms were selected for the experiment. Figure 4 shows a block diagram for both applications. The following items have been noticed on both applications:

1. Need for user interface to allow the creation of scheduling data.
2. Storage mechanism for scheduling data.
3. Performance calculation of objective functions.

4. Classes for data validation to make sure that the data is consistent and can be used.

5. User interface mechanism for presenting the results to users.

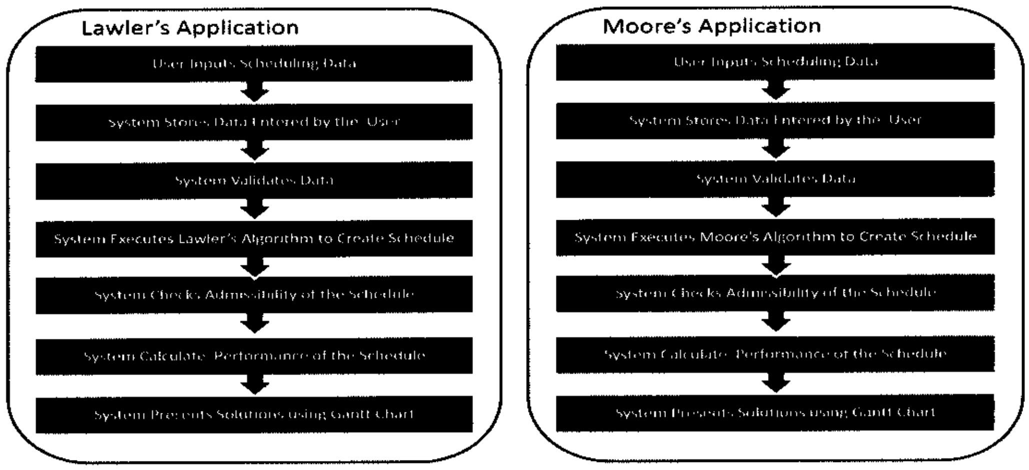

Lawler's Application
User Inputs Scheduling Data
System Stores Data Entered by the User
System Validates Data
System Executes Lawler's Algorithm to Create Schedule
System Checks Admissibility of the Schedule
System Calculate Performance of the Schedule
System Presents Solutions using Gantt Chart
Moore's Application
User Inputs Scheduling Data
System Stores Data Entered by the User
System Validates Data
System Executes Moore's Algorithm to Create Schedule
System Checks Admissibility of the Schedule
System Calculate Performance of the Schedule
System Presents Solutions using Gantt Chart

FIGURE 4. Block diagram for two distinct scheduling applications.

After the first analysis for the identification of potential business blocks that could be implemented at framework level, a second study has been done. This time, scheduling algorithms themselves have been the object of the analysis. The motivation for this study was to identify common objects and operations that are present in different scheduling algorithms and therefore can be abstracted to a framework level in order to be shared between many different algorithms.

The following algorithms and dispatching rules have been used for this experiment:

1. Lawler's algorithm.
2. Moore's algorithm.
3. Johnson's algorithm.
4. FAM-EDD.
5. FAM-MS.

This iteration has been very helpful to demonstrate that scheduling algorithms, in spite of being very different from each other, still share similarities at certain steps in their logics. An illustrative example was found when analyzing the dispatching rules. All dispatching rules analyzed during this study have demonstrated the need for sorting orders based on a given criteria. This makes it clear, without the need of formal proof, that any support, at framework level, for sorting orders or operations based on different criteria would benefit the implementation of any dispatching rule. Another proof of the existence of similarities can be illustrated with certain operations such as "Find Order with Maximum Processing Time" or "Find Order with Earliest Due Date." Those are frequently found in many different algorithms.

The second and perhaps the most important outcome of this second analysis was the identification of a common data model. Every algorithm evolved around a well defined set of entities: orders (jobs), operations, machines (resources). That came to no surprise since the majority of scheduling literature uses those objects to describe most scheduling algorithms. What was important was the identification of the need for classes that could allow collection of objects to be stored and easily retrieved. Those were later implemented in the framework as lists and collections of orders, lists and collections of operations and etc.

It is important to point out that this second iteration is by no means completed. A real implementation of a robust framework would require this analysis to be performed on a much broader number of algorithms to make sure it is providing support for implementing a large range of distinct scheduling algorithms. At least one algorithm per environment configuration should be used to guarantee that the operations and properties found are really common throughout several algorithms and can be securely abstracted to base classes at framework level.

The following list presents the new candidate items found after the second analysis:

1. A common data model composed mainly of orders, operations, work centers and resources has been noticed on all algorithms and rules analyzed.

2. A certain number of operations, such as "sort orders by due date," "sort orders by processing time," "find order with greatest processing time."

The last two items added to the framework were actually identified after the development of the framework had already been started. During the initial development stages of the framework, it has been noticed the need for reporting inconsistencies during data validation and also the need for logging information for debug purpose. At that point, a decision was taken to add to the framework support for logging and status notification using exception. Therefore, the last two business blocks added to the framework were:

1. Support for logging debug information.
2. Support for exception.

The focus of this work is to provide a starting point for implementing new scheduling algorithms; therefore it will take advantage of any open source code available that might add functionality to the framework. The implementation will be done using the Microsoft . Net platform.

Framework Namespaces

The list of items cited above has been divided in several namespaces. The reasoning behind this approach is to have a different set of functionality per namespace. The break down helps users of the framework to easily find classes or objects since all items related to the same block of functionality are located in the same namespace. Moreover, having different namespaces helps to avoid extremely large projects which in turn helps the maintainability of the code. Splitting the code in different namespace based on functionality also helps to minimize coupling and it is more likely to be coherent. The coupling is minimized due to the fact that the developer can easily see when a dependency is introduced in the code since a reference to a difference namespace has to be added. The coherence is increased because separated namespaces deals with a unique block of functionality.

The following namespaces have been used to build the scheduling framework. These set of namespace has been defined based on the main blocks identified during the iteration described earlier:

1. SchEdu. Framework. Algorithm.

2. SchEdu. Framework. Algorithm. Rules.
3. SchEdu. Framework. Constraint.
4. SchEdu. Framework. DataModel.
5. SchEdu. Framework. Exceptions.
6. SchEdu. Framework. Logging.
7. SchEdu. Framework. Objective.
8. SchEdu. Framework. Validators.
9. SchEdu. Framework. Win.

SchEdu. Framework. Algorithm Namespace

Classes in the Algorithm namespace are suppose to be base classes for implementing different scheduling algorithms. Figure 5 shows the most important classes available in the Algorithm namespace.

IAlgorithm is an interface implemented by every algorithm class in the framework. The IAlgorithm interface allows client code to utilize any class that implements the IAlgorithm interface. That was done to allow client code to program to interface not to an implementation as recommended by (11). The same approach has been used in other several other places throughout the Scheduling Framework.

The Algorithm class provides functionally that are common between all different algorithms regardless if the problem being address belongs to single machine category, or parallel machine category, or etc. Therefore, operations not tight to any specific type of environment configuration have been implemented at Algorithm class. Any algorithm, regardless if it was built for a single machine or parallel machines can benefit of those operations present at Algorithm class since Algorithm is suppose to be the base class for any algorithm implemented using the scheduling framework.

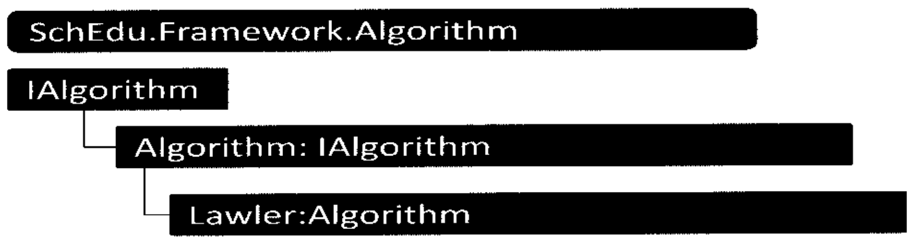


FIGURE 5. Algorithm namespace.

The design of the Algorithm class has been based on the Template Pattern. The Template Method design pattern defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure (11). The Template method has been selected due to the fact that the executing sequence of all scheduling algorithm is similar. That makes possible for the framework to provide a skeleton and let the concrete implementation for a given scheduling algorithm to provide implementation for specific methods.

An abstract implementation for Lawler's algorithm (12) has also been added to the framework in the Algorithm's namespace. The reasoning behind the decision of adding support for a specific type of algorithm is based on the nature of the Lawler's algorithm. Lawler's can be seeing as a generic algorithm that solves $1|prec|\ f_{max}$ where

f_{max} is any function of the completion time that has a monotone non-decreasing behavior. Lateness and tardiness can be good examples of the Lawler's reusability. If f_{max} is defined by $fmax = \max_{i=1..n}(Ci - di)$, Lawler's can be used to minimizing f_{max} and in turn minimize lateness. A similar approach can be used for tardiness by defining $fmax = \max_{i=1..n}(\max(Ci - di, 0))$

<u>SchEdu. Framework. Algorithm. Rules Namespace</u>

The Rules namespace is a sub-namespace under the Algorithm namespace (Figure 6). The Rules namespace has been used for implementing support for a variety of different dispatching rules such as FAM-EDD (Fist Available Machine Earliest Due Date) and FAM-LPT (First Available machine Longest Processing Time). The decision to create the sub-namespace has been taken due to the fact that dispatching rules are commonly seeing as special types of scheduling algorithms. Moreover, dispatching rules are often found as a step in the implementation of other scheduling algorithms.

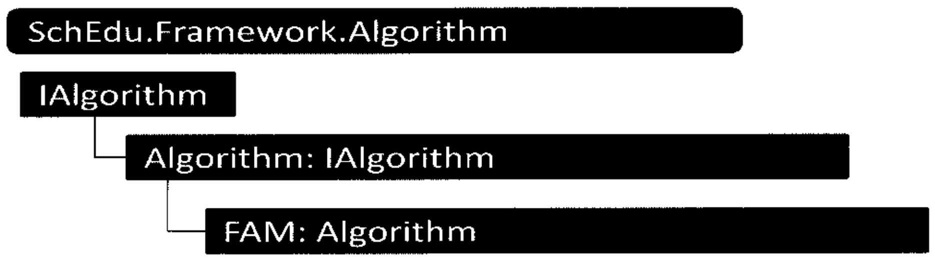


FIGURE 6. Simplified view of the rules namespace.

The Rules namespace provides also a class called FAM for supporting the implementation of dynamic rules. The FAM class can be used to implement several different dynamic rules by implementing a single abstract method. The Template Method design pattern has been used similarly to what was done for the Algorithm abstract class. A method is invoked every time a list of operations is available to be scheduled. Developers have just to apply the desired dispatching rules against the list of operations provided by the base class. This approach has greatly simplified the development of different dynamic rules.

SchEdu. Framework. Constraint Namespace

The Constraint namespace provides classes for validating whether the schedule violates any constraints. A common type of constraint is the finish date constraint. If any order in the schedule ends after the assigned finish date, the entire schedule is considered inactive. Active schedules are those which don't violate any constraint present in the problem definition. This namespace provides support for adding new constraint validations. Users can use these constraints to validate the result of any scheduling algorithm generated by the framework. Figure 7 provides a simplified view of the Constraint framework.

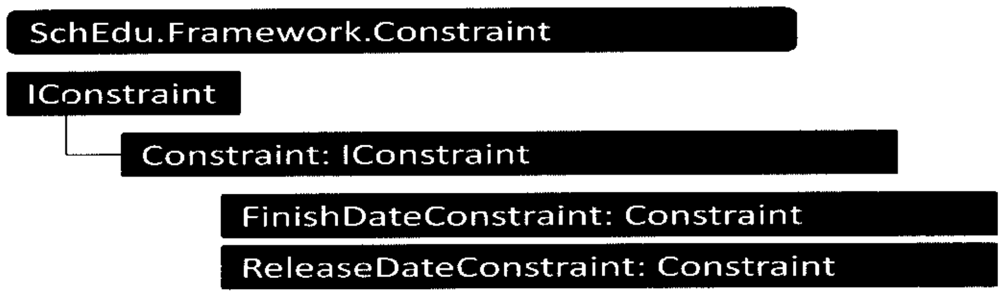

SchEdu.Framework.Constraint
IConstraint
Constraint: IConstraint
FinishDateConstraint: Constraint
ReleaseDateConstraint: Constraint

FIGURE 7. Constraint namespace.

SchEdu. Framework. DataModel Namespace

The designing of a data model is a crucial task on any software development. Developers spend a large amount of time trying to create a data model that can handle all the necessary data for their algorithms. The data model describes how the data necessary for a given algorithm could be represented using different classes. Data models are usually described in terms of data elements and their relationships. Object orientation has been the most popular technique used by computer engineers to build their data models on the last couple of years. Abstraction, inheritance, composition and many other features provided by object oriented languages has been extensively used and have greatly helped the creation of data models.

There are several qualities attributes such as simplicity, extensibility, and performance, which are in the mind of developers while building data models (13). A data model should be simple to use and to understand. Provide an easy way to be extended in order to be adapted to new functionalities and requirements. Finally, it should perform well when applied to the range of scenarios to which it was designed for.

The DataModel namespace provides data classes for handling the main entities found in scheduling applications. The DataModel can be seen as a database centric namespace. Classes for storing orders, operations, work centers, etc have been defined in the DataModel namespace. This namespace provides also several other data classes such as order collections or dependency collection that can be very helpful while implementing any type of logic.

The mains class in the DataModel namespace is the SchEduDataSet. The SchEduDataSet class contains data tables for orders, operations, work centers, resources, dependencies, allocations, etc. The data model supports serialization[4] and it is data base independent which allows it to be stored on different commercial data base software available in the market such as Microsoft SQL, Oracle, etc. The data model supports also a set of constraints such as foreign keys and unique keys which provides integrity to the schedule data. This is very important because it removes the responsibility of checking the integrity of the date from the developer.

The SchEduDataSet class can be easily extended by adding new tables. It is important for the framework to allow users to add new tables in other to accommodate any extra information that might be required by their own algorithms. Figure 8 provides

[4] Serialization: is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file, a memory buffer, or transmitted across a network connection link to be "resurrected" later in the same or another computer environment.

a simplified view of the SchEduDataSet class with all the existing entities and their relationships.

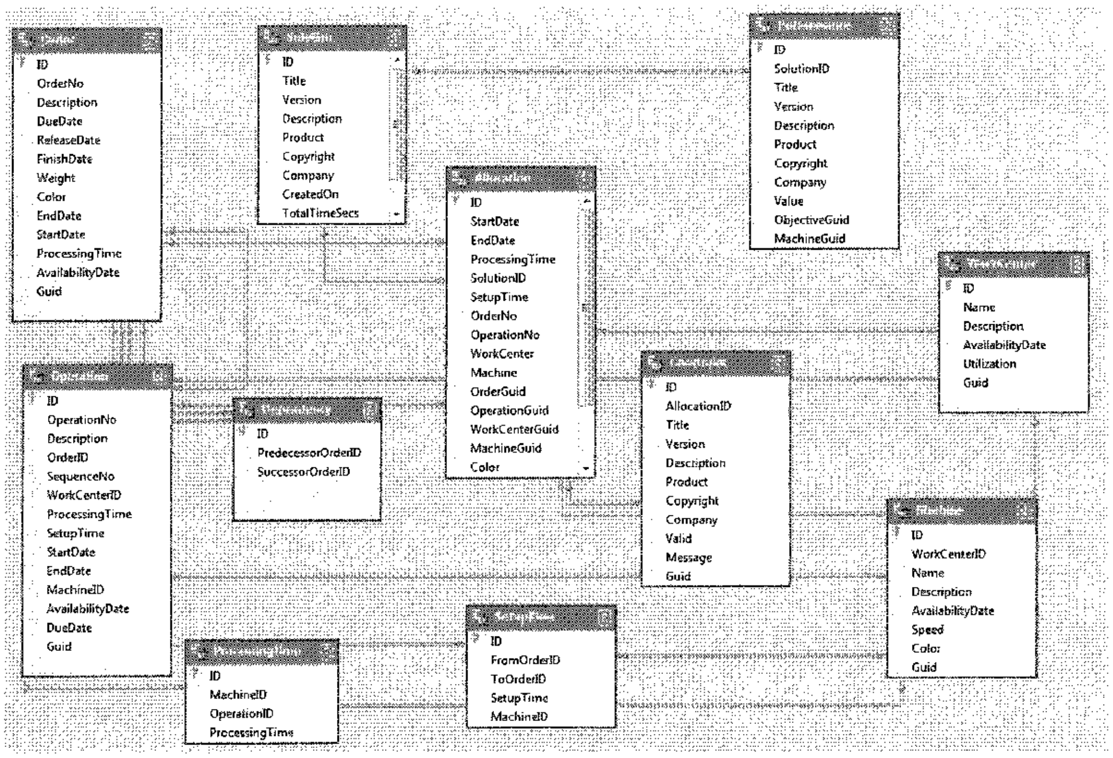


FIGURE 8. SchEduDataSet class.

The SchEduDataSet class is composed of eleven tables: Order, Operation, WorkCenter, Machine, Dependency, SetupTime, ProcessingTime, Solution, Allocation, Performance and Constraint.

These eleven tables could be logically divided in three different categories:

1. Scheduling Data.

2. Environment Data.

3. Solution Data.

The scheduling data group provides the data used by scheduling algorithms to generate possible solutions. Tables Order, Operation, Dependency, ProcessingTime and SetupTime belong to it.

The second logical group, environment data, is composed by WorkCenter and Machine tables. This group describes the environment where the orders will be scheduled.

Finally the last groups of tables provide storage for solutions that are generated by the execution of scheduling algorithms. It also contains table for performance calculation and constraint violation. Tables available in the data model namespace are explained below.

Order Table

The Order table is a collection of OrderRow objects (Figure 9). Each instance of OrderRow represents an order to be scheduled. The OrderRow object holds information regarding the order to be scheduled. The OrderRow has several properties such as start date, end date, due date, weight and etc. OrderRow is a master record for the operation table. An order is composed of operations.

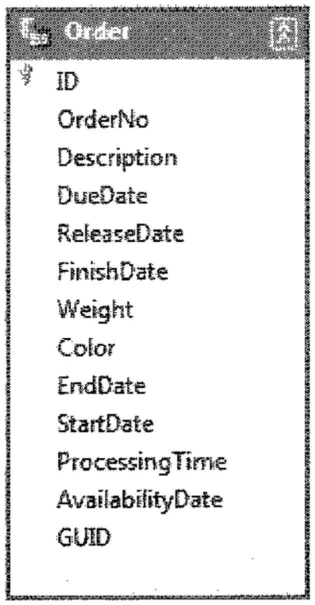


FIGURE 9. Order entity.

Operation Table

The Operation table is a collection of OperationRow objects (Figure 10). Each instance of OperationRow represents a processing required by a given order at a given work center. In manufacturing environment, orders are often divided in several operations. Each operation represents a certain processing time required by the order at a given work center. The Operation table is generic enough to support different types of problems where:

1. Operation sequence is a constraint and must be followed.
2. All orders have the same sequence of operations (Flow shop).
3. Orders have different sequences of operations (Job shop).

4. Operations are independent and no rigid sequence must be followed (Open shop).

5. Operations can be executed by any resource available at the work center (Flexible shops).

A foreign key constraint has been created between the Order table and the Operation Table to guarantee that no operation would exist without an order. The constraint has been created with the "cascade delete" option to make sure that all operations are deleted once its parent order has been deleted.

Operations are also called jobs in the scheduling literature. The foremost objective of any scheduling algorithm is to assign a start date for all operations of a scheduling problem. In single machine (single work center) problems, orders are often called jobs. The reason behind that is simple. In single machine problems, it is implicit that an order must be executed in the only work center available in the problem, therefore it is implicit that every order has a single operation to be executed at the work center. In that case, the processing time is usually defined at order level and not at operation level. In order to create a more homogeneous data model, the scheduling framework requires every order to have at least one operation and the processing time is always defined at operation level.

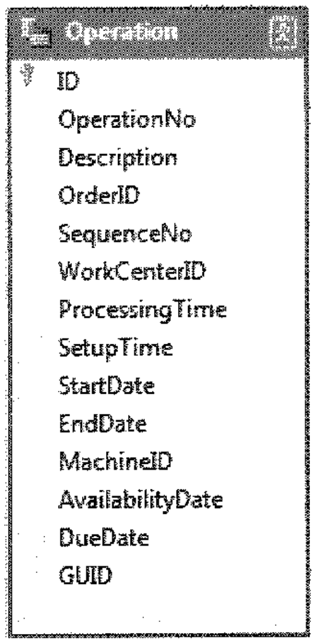


FIGURE 10. Operation entity.

Dependency Table

The Dependency table is a collection of DependencyRow objects (Figure 11). Several algorithms deal with dependencies. When an order is dependent on another order, it cannot be started while the other order has not finished. Dependencies are a very common type of constraint in manufacturing environments. It is common in complex manufacturing processes to break an order for a finish good in several sub-assembly orders and create dependencies these sub assembly orders.

Dependencies are usually grouped in 3 different types (2). If a job has at most one predecessor and at most one successor, the dependency constraint is called chain. When the jobs have at most one successor, it is called 'in tree' and when jobs have at most one predecessor, ‘out tree’. The dependency table supports all those 3 types of dependencies.

The most common data model used to represent dependencies is a square bi-dimensional matrix where the number of columns and rows is basically the number of jobs to be scheduled. One in a matrix cell (i, j) indicates that Order_i is dependent on Order_j or Order_i cannot start while Order_j has not finished (Table 4).

TABLE 4. Common Dependency Representation

	Order1	Order2	Order3
Order1		1	
Order2			1
Order3			

The major drawback in representing dependencies with matrix is that relational databases do not support matrix. Therefore, that would represent a challenge for the framework if the data had to be stored in a relational database. A dependency collection will be used instead of a bi-dimensional matrix.

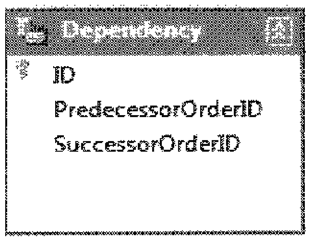


FIGURE 11. Dependency entity.

ProcessingTime Table

The ProcessingTime table is a collection of ProcessingTimeRow objects (Figure 12). The ProcessingTime table is used by problems with unrelated machines.

The data model supports 3 categories of resource environment regarding processing time:

1. Single machines or identical machine: Operations have processing requirement and that is the only information needed since only one type of machine is present.

2. Uniform machines: Operations have processing requirement and machines have speeds. The processing time is depended on the machine where the order is executed. If an Order_i is processed on Machine_J then the processing time is Order_i ProcessingTime / Machine_J Speed.

3. Unrelated Machines: The last and more generic case is when dealing with unrelated machines. Operations have a processing time for every machine where it is allowed to be executed.

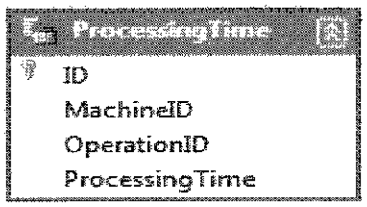


FIGURE 12. Processing time entity.

SetupTime Table

The SetupTime table is a collection of SetupTimeRow objects (Figure 13). Each record in the SetupTime table represents a time required to switch between two orders on a given machine. Setup times are dependent on three variables: current order, next order and machine. A certain amount of time is usually required to switch between two orders on a given machine. This time has a very significant impact in the total duration of the schedule. Therefore, it is quite common to see scheduling algorithms where the main objective is related to setup times. Since flexible flow shops and flexible job shops might have more than one machine at a given work center, the data model supports setup times at machine level. Most manufacturing environments don't have more than one machine per work center and even if they did, most probably they would be similar machines. Regardless of that, the data model allows setup times at machine level.

The next two tables, work center and machine, describe the environment where the orders will be executed.

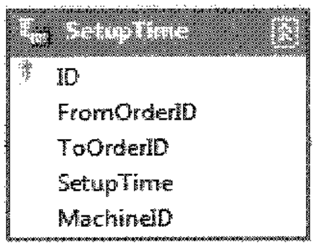


FIGURE 13. Setup time entity.

WorkCenter Table

The WorkCenter table is a collection of WorkCenterRow objects (Figure 14). Work centers are often used to group resources such as machines, people, etc. For single machine problems, the work center itself is usually the resource to be allocated. However, the scheduling framework requires every work center to have at least one machine. The reason is similar to the one used for the operation model: homogeneity.

The scheduling data model also supports environments where work centers have more than one machine. The complexity of this type of problems is much bigger than those where work centers have just a single machine. Scheduling algorithms have not only to assign dates to operations but also to choose a machine. These problems are often called "scheduling with assignment."

Machine Table

The machine table is a collection of MachineRow objects (Figure 15). Machines belong to work centers, therefore a foreign key between the WorkCenter table and the Machine table has been created. The foreign key constraint between both tables has been defined with "delete cascade" options. By doing that, the data model guarantees that a machine would not exist without is parent work center. A property for machine speed has been added to the machine entity. That is important for scheduling problems where the time requirement for an operation at a given work center is dependent on the speed of the machine.

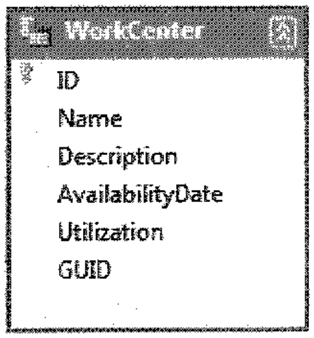


FIGURE 14. Work center entity.

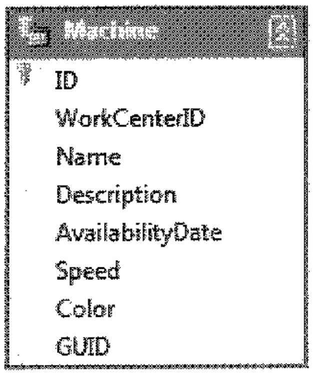


FIGURE 15. Machine entity.

The last group of tables, the solution tables, was created to store the results of the execution of a scheduling algorithm.

Solution Table

The Solution table is a collection of SolutionRow objects (Figure 16). Every time an algorithm is executed by the framework, a new solution record is created in the Solution table. The Solution table holds information regarding the algorithm used for the solution, the execution time of the algorithm, and other important information. A solution is composed of allocations.

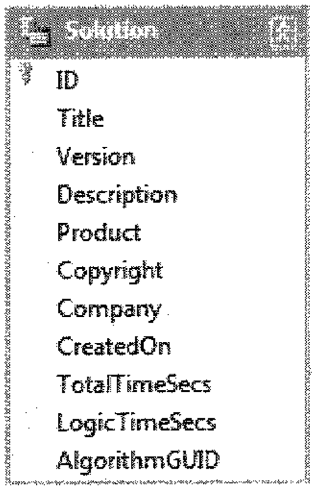


FIGURE 16. Solution entity.

The scheduling framework allows multiple solutions to be store for a given problem. This is an important feature of the framework because it allows different solutions to be compared.

Allocation Table

The allocation table is a collection of AllocationRow records (Figure 17). Each record in the allocation table represents a machine allocation against an operation during a given period of time. The allocation table is the source of data used to display a Gantt chart.

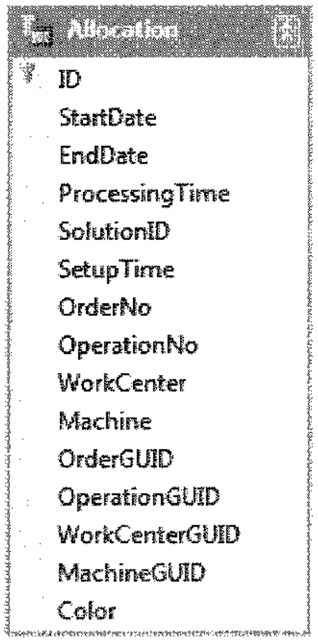


FIGURE 17. Allocation entity.

It is important to notice that the allocation table does not have any references to order, operation, work center, machine and etc. The decision to avoid references to other tables was done to allow changes to the problem without affecting solutions that were previously generated. It allows developers to generate a solution for a given problem, make changes to the problem, generate a new solution and finally compare both solutions. A developer could be analyzing how his algorithm behaves when used against the same orders with a different environment configuration.

Performance Table

The performance table is a collection of PerformanceRow records (Figure 18). Each row stores the result of an objective function calculated for the solutions. The table stores not only the value of the objective function but also certain data elements that describe the objective class used for the calculation. Those elements serve as documentation for the record. Storing performance information per solution had an important implication. This model allows the comparison between objectives calculated for different solutions.

Constraint Table

The constraint table is a collection of ContraintRow records (Figure 19). Each record represents the status of an allocation in regards a given constraint. Constraints are checked for every allocation and one single allocation could be violation more than one constraint.

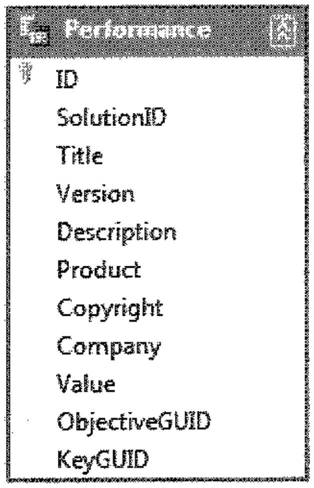


FIGURE 18. Performance entity.

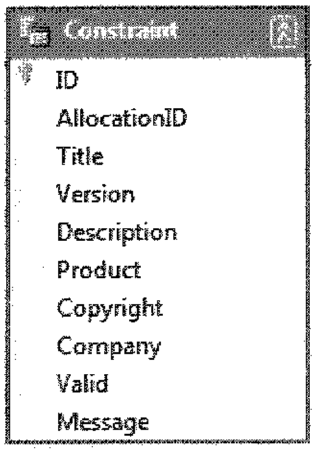


FIGURE 19. Constraint entity.

Besides the data tables used to store the problem information and the solutions generated by the different algorithms, the DataModel namespace also provides several other data classes that are of great help while implementing scheduling algorithms.

Figure 20 shows the major data classes available in the DataModel namespace. Each of those classes provides functionality that can be helpful while implementing scheduling algorithms.

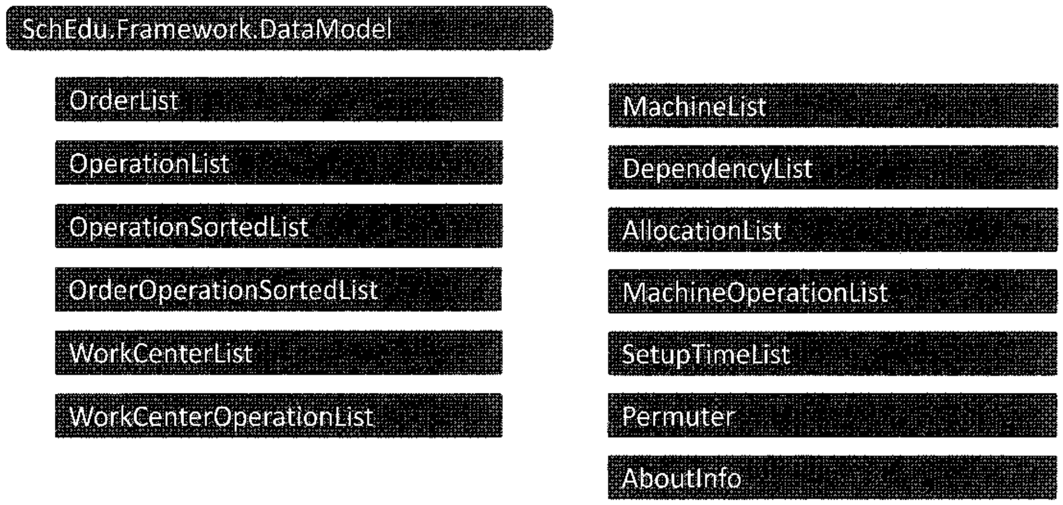


FIGURE 20. Data classes available in the DataModel namespace.

SchEdu. Framework. Exceptions Namespace

The Exceptions namespace implements all exceptions that are thrown by the Scheduling Framework. Validators, Constraints, Algorithms and other classes, they all use exceptions defined in the Exception namespace in order to report inconsistencies during the execution of the code. Figure 21 provides a simplified view of the Exceptions namespace.

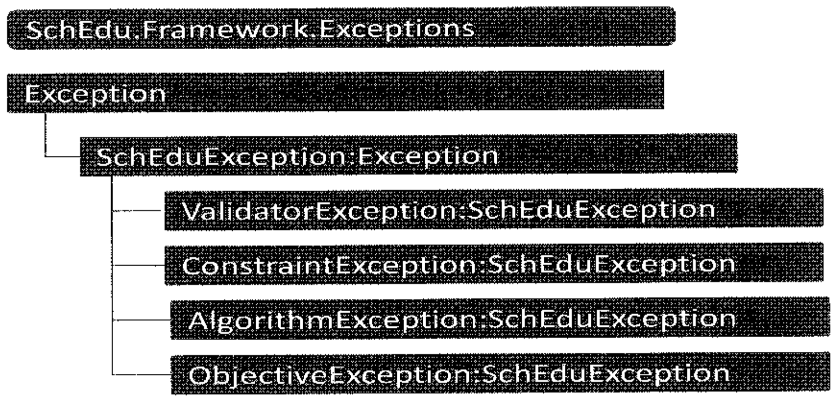

SchEdu.Framework.Exceptions
Exception
SchEduException:Exception
ValidatorException:SchEduException
ConstraintException:SchEduException
AlgorithmException:SchEduException
ObjectiveException:SchEduException

FIGURE 21. Exceptions namespace.

There are currently 4 base classes for exceptions. Other new exception classes could be created by developers by deriving from the existing classes. The four classes currently available have been defined based on the most common places where developers will make use of exceptions in their implementations. Developers will be mainly creating new algorithms, or new validations, or new constraints or new objectives. Therefore, an exception class for each of those development places has been added to the framework.

SchEdu. Framework. Logging Namespace

The Logging framework provides the Logger class. The Logger class is a helper class that allows any client code to create logging information in several different formats. The logger class is a wrapper class created on top of Log4Net (Log for Net). Log4Net is an implementation of the well known Log4Java (Log for Java) framework for the Microsoft . Net environment. All base classes in the Scheduling Framework have a reference to the Logger class to provide a seamlessly mechanism for logging information and to allow any class that inherits from base classes to also benefit from the logging support. Figure 22 shows the Logging Framework simplified view.

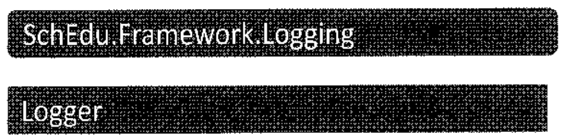


FIGURE 22. Logging namespace.

The integration of Log4Net to the scheduling framework has made available such a powerful set of logging features. All features can be easily configured by changing just the framework configuration file.

SchEdu. Framework. Objective Namespace

The Scheduling Framework provides a certain number of classes to calculate the performance of a schedule. The performance is calculated based on different objective functions. The Objective namespace provides classes for calculating the output of the most common objective functions found in scheduling problems. Figure 23 depicts the objective classes available in the Objective namespace.

The framework supports calculation of objectives for the entire solution or per machine as well. Certain objective classes such as MachineUtilizationTime or MachineUtilizationPercentage calculates values per machine in the solution. The objective namespace provides a base class for creating new performance calculations.

Developers can easily inherit from the Objective class in order to develop their new performance calculation. The Template Method design pattern has been used to design the Objective base class. The approach was similar to the one described for the Algorithm class. Objective functions created by the developers can be added to the framework by just modifying the framework configuration file.

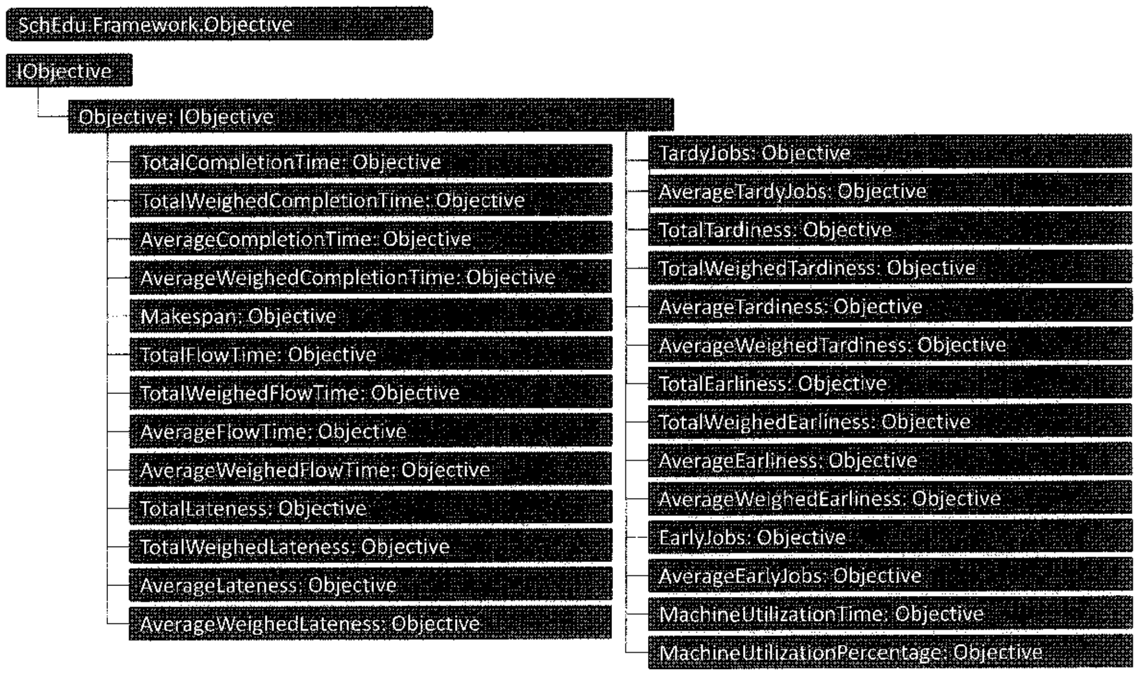

SchEdu.Framework.Objective
IObjective
Objective: IObjective
TotalCompletionTime: Objective
TotalWeighedCompletionTime: Objective
AverageCompletionTime: Objective
AverageWeighedCompletionTime: Objective
Makespan: Objective
TotalFlowTime: Objective
TotalWeighedFlowTime: Objective
AverageFlowTime: Objective
AverageWeighedFlowTime: Objective
TotalLateness: Objective
TotalWeighedLateness: Objective
AverageLateness: Objective
AverageWeighedLateness: Objective
TardyJobs: Objective
AverageTardyJobs: Objective
TotalTardiness: Objective
TotalWeighedTardiness: Objective
AverageTardiness: Objective
AverageWeighedTardiness: Objective
TotalEarliness: Objective
TotalWeighedEarliness: Objective
AverageEarliness: Objective
AverageWeighedEarliness: Objective
EarlyJobs: Objective
AverageEarlyJobs: Objective
MachineUtilizationTime: Objective
MachineUtilizationPercentage: Objective

FIGURE 23. Objective namespace.

SchEdu. Framework. Validators Namespace

The Validators Framework provides Validators classes that can be used by other classes in order to validate that certain conditions are met. The namespace has classes to validate conditions such as due date, processing time or that an environment complies with single machine configuration. Besides validating the source data, the framework provides also classes to validate that the solution generated by an algorithm is valid as well. Figure 24 depicts the Validators namespace.

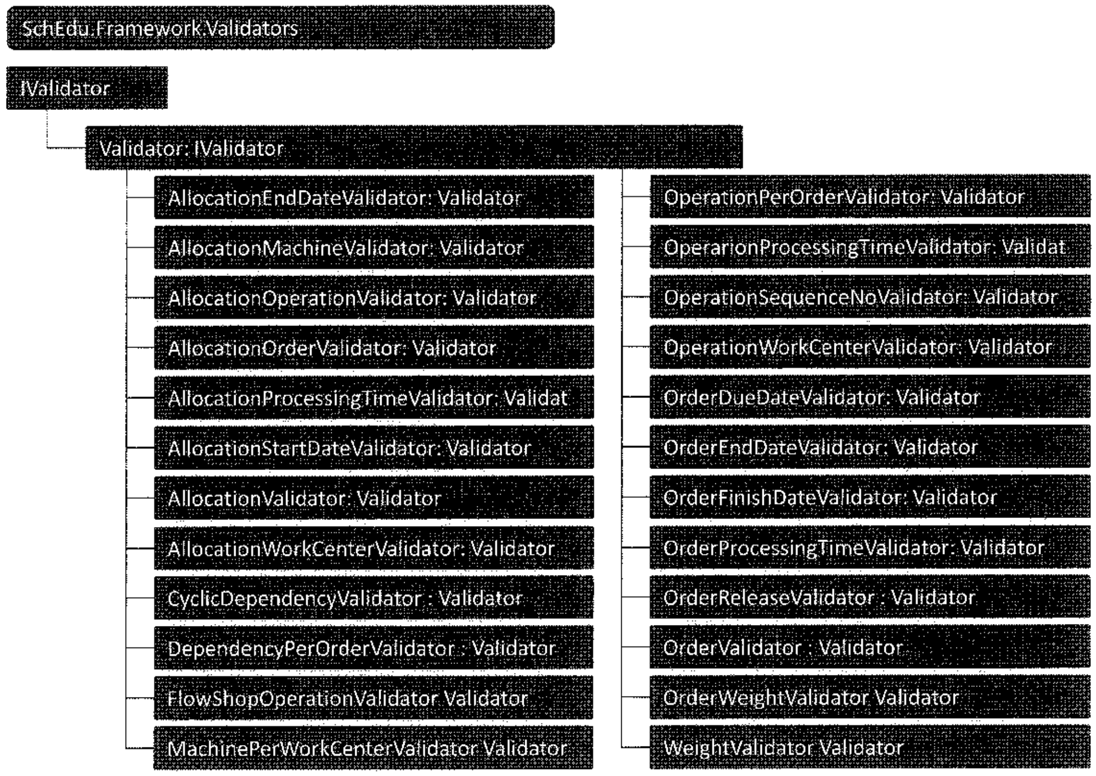

SchEdu.Framework.Validators
IValidator
Validator: IValidator
AllocationEndDateValidator: Validator
AllocationMachineValidator: Validator
AllocationOperationValidator: Validator
AllocationOrderValidator: Validator
AllocationProcessingTimeValidator: Validat
AllocationStartDateValidator: Validator
AllocationValidator: Validator
AllocationWorkCenterValidator: Validator
CyclicDependencyValidator : Validator
DependencyPerOrderValidator : Validator
FlowShopOperationValidator Validator
MachinePerWorkCenterValidator Validator
OperationPerOrderValidator: Validator
OperarionProcessingTimeValidator: Validat
OperationSequenceNoValidator: Validator
OperationWorkCenterValidator: Validator
OrderDueDateValidator: Validator
OrderEndDateValidator: Validator
OrderFinishDateValidator: Validator
OrderProcessingTimeValidator: Validator
OrderReleaseValidator : Validator
OrderValidator : Validator
OrderWeightValidator Validator
WeightValidator Validator

FIGURE 24. Validators namespace.

The Validators namespace also provides an extensibility mechanism similar to the Algorithm, Objective and Constraint namespaces. A base class is available for supporting the development of new algorithms. Developers can inherit from the

Validator base class to create new Validators to guarantee the integrity of the data required by their algorithms.

SchEdu. Framework. Win Namespace

The Win namespace provides support adding user interface capabilities to any application that uses the scheduling framework. Figure 25 shows the main classes available in the Win namespace.

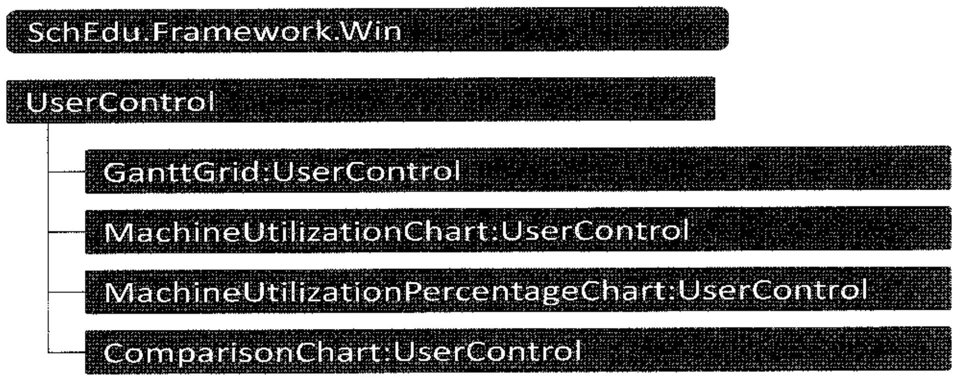

SchEdu.Framework.Win
UserControl
GanttGrid:UserControl
MachineUtilizationChart:UserControl
MachineUtilizationPercentageChart:UserControl
ComparisonChart:UserControl

FIGURE 25. Win namespace.

There are four main classes in the Win namespace: GanttGrid, MachineUtilizationChart, MachineUtilizationPercentageChart and ComparisonChart. The GanttGrid class provides users a mechanism to display the result of their algorithms on a Gantt chart format. The class is able to display Order Gantt and a Machine Gantt. Figure 26 shows data being displayed on a Job Gantt chart using the classes in the Win namespace.

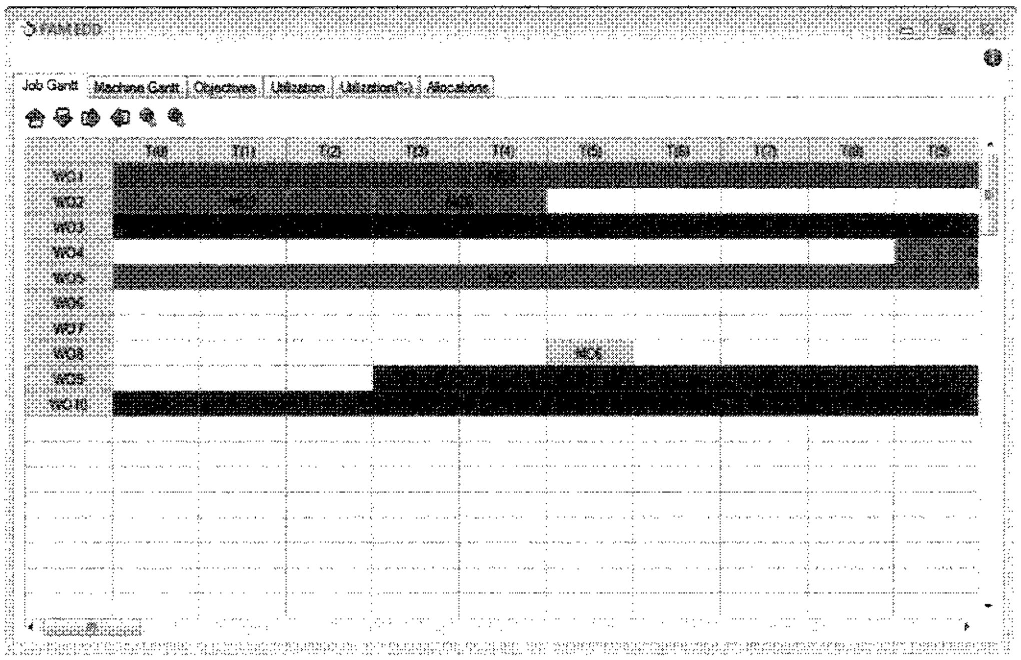


FIGURE 26. Gantt chart component available in the Win namespace.

The Gantt chart on Figure 26 shows the result of the FAM-EDD applied to a scheduled data that was generated ran randomly using the generated data also provided by the framework. The job Gantt shows the machine allocation required by the solution.

The MachineUtilization classes allow developers to visualize the machine utilization required by the solution produced by their algorithms. There are two types of machine utilization chart: MachineUtilizationChart and MachineUtilizationPercentageChart. The first, allows the developers to see the occupancy of machines based on the timeline of the schedule. Figure 27 shows the occupancy of machines during the duration of a given schedule. The occupancy was generated by the FAM-EDD algorithm applied to a certain number of orders. The next chart, Figure 28, displays again the occupancy of the same machines but as a percentage of the timespan.

The last class available in the Win namespace, the ComparisonChart, allows developers to compare the objective of multiple solutions on a same chart. Figure 29 shows two solutions, FAM-EDD and FAM-LPT, being compared in by the ComparisonChart. Those two algorithms were applied to the same data.

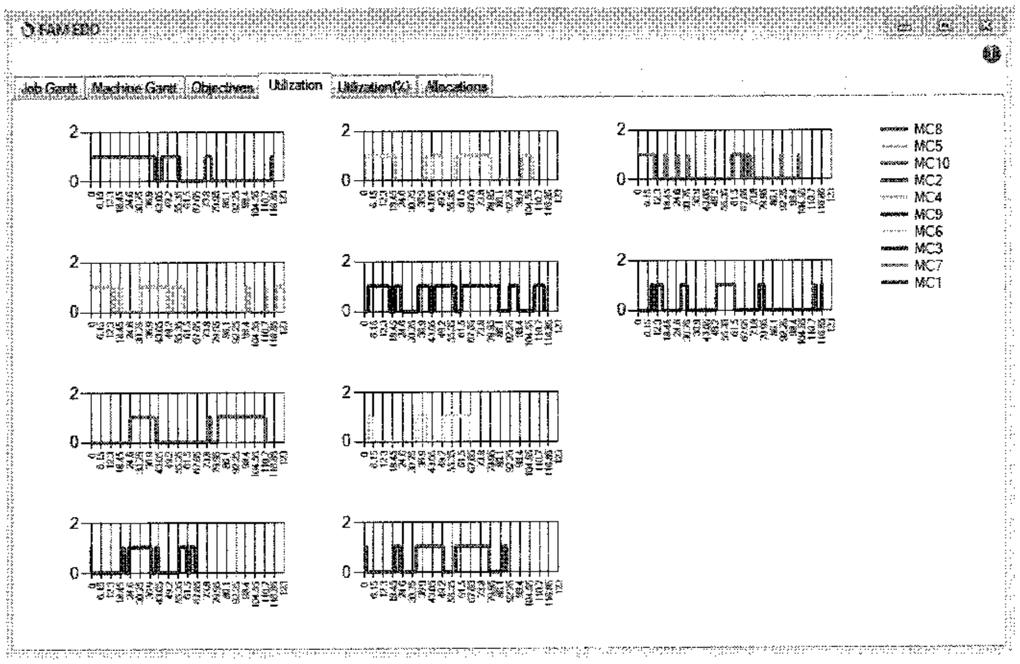


FIGURE 27. Machine utilization chart.

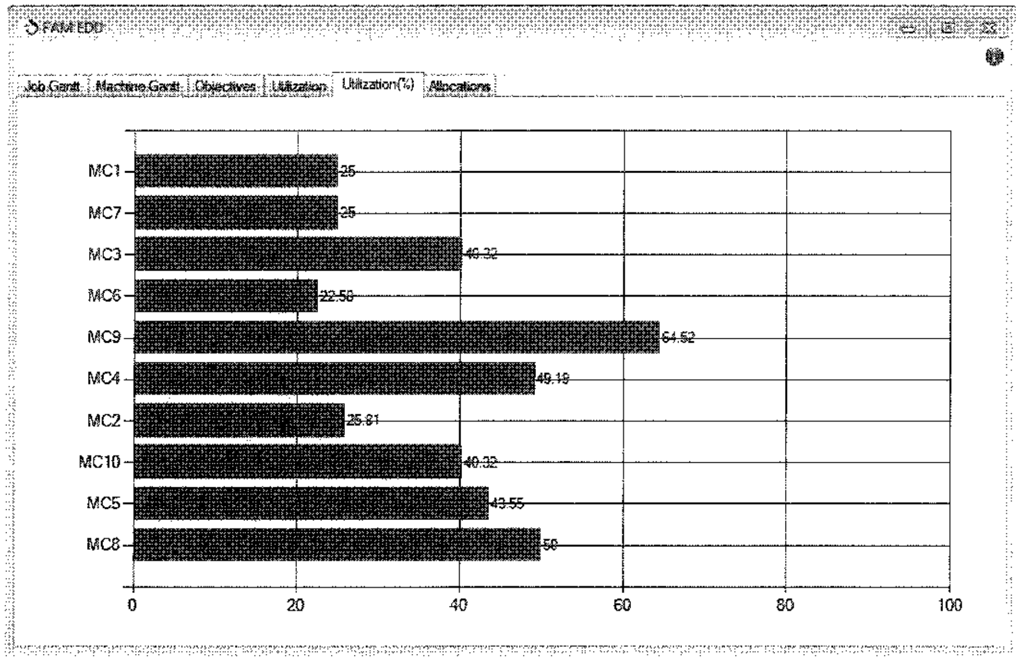


FIGURE 28. Machine utilization percentage chart.

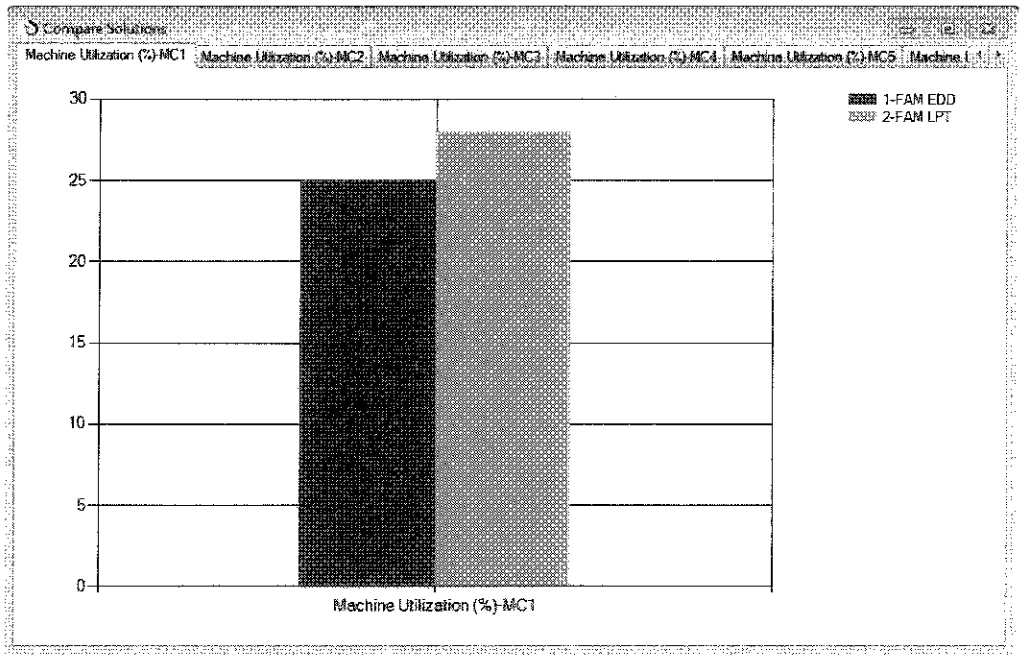


FIGURE 29. Comparison chart.

SchEdu. WinApp Application

A Windows application has also been developed as part of the framework. The SchEdu. WinApp application is suppose to provide developers of scheduling algorithms a mechanism for creating their testing data and visualizing the results of their implementations. This application has been developed using a plug-in model which allows new algorithms, new objectives and new constraints to be added to the application by simply modifying its configuration files.

Developers can just modify the configuration file to load their algorithms. Moreover, the SchEdu. WinApp application also allows objective functions to be loaded dynamically. This flexibility allows developers to also create new objective functions and view the results in the WinApp screen without any extra effort. The following fragment of the configuration files shows how the Moore algorithm implemented in a separated project has been loaded into the WinApp.

```
<algorithmMenuItems>
  <algorithmMenuItem name="Single Machine">
    <algorithmMenuItems>
     <algorithmMenuItem name="Moore" class="Moore. MooreAlgorithm" assembly="Moore,
Version=10. 0. 0, Culture=neutral, PublicKeyToken=d0d9963d53fac747"/>
```

The results of these lines in the configuration files is a sub-menu item called "Single Machine" in the algorithm's menu followed by an menu item called "Moore" under the "Single Machine sub-menu." This model allows developers to build sub-menus to organize their algorithms under different categories.

Similarly to algorithms, objective functions can be loaded dynamically using the WinApp configuration file. The code below shows how the Total Completion Time objective function has been loaded.

```
 <objectiveItems>
   <objectiveItem name="Total Completion Time" class="SchEdu. Framework. Objective.
TotalCompletionTime" assembly="SchEdu. Framework. Objective, Version=1. 0. 0. 0, Culture=neutral,
PublicKeyToken=d0d9963d53fac747"/>
```

The most important functionality provided by the SchEdu. WinApp is its capabilities to create test data and supply the data to any algorithm that has been specified in the configuration file.

The application provides four different screens for entering scheduling data (Figures 30, 31, 32, and 33): environment, orders, dependencies and setup times. The Environment screen is used to enter data regarding the shop floor configuration. The screen has two grids: work centers and machines. Developers can enter their work center configuration together with all machines available on each work center. The Orders screen allows developers to create their test orders to be used in their algorithms. The screen is also composed of two grids: orders and operations. Users can supply operations for their orders. The Dependencies screen provides users a mechanism to configure dependencies between orders. Finally the Setup Time screen allows developers to enter the setup time required to switch between two orders on a given work center.

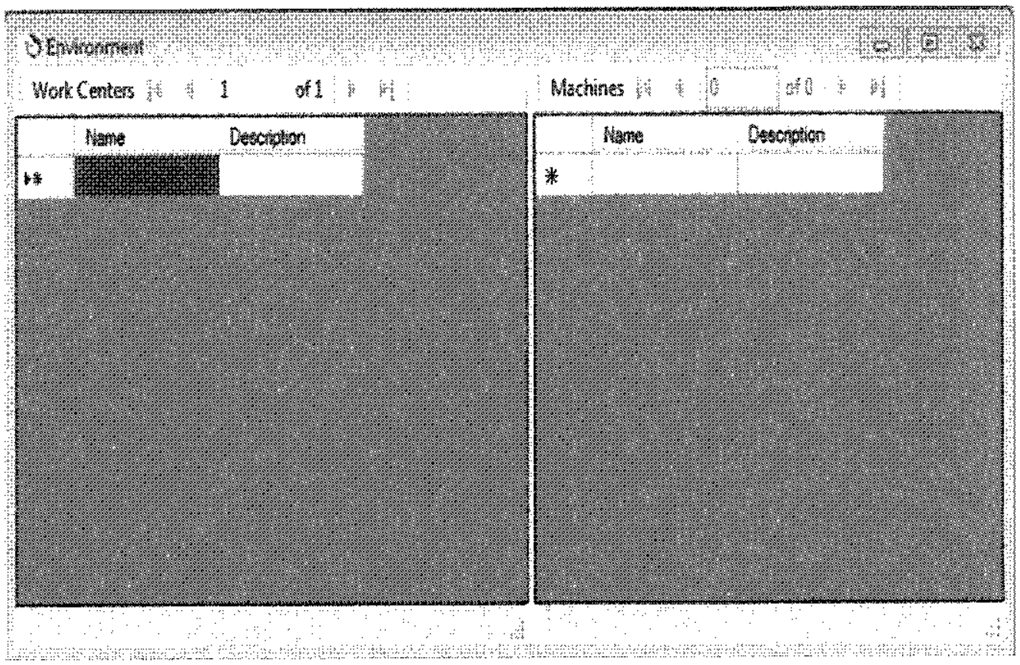


FIGURE 30. Environment screen.

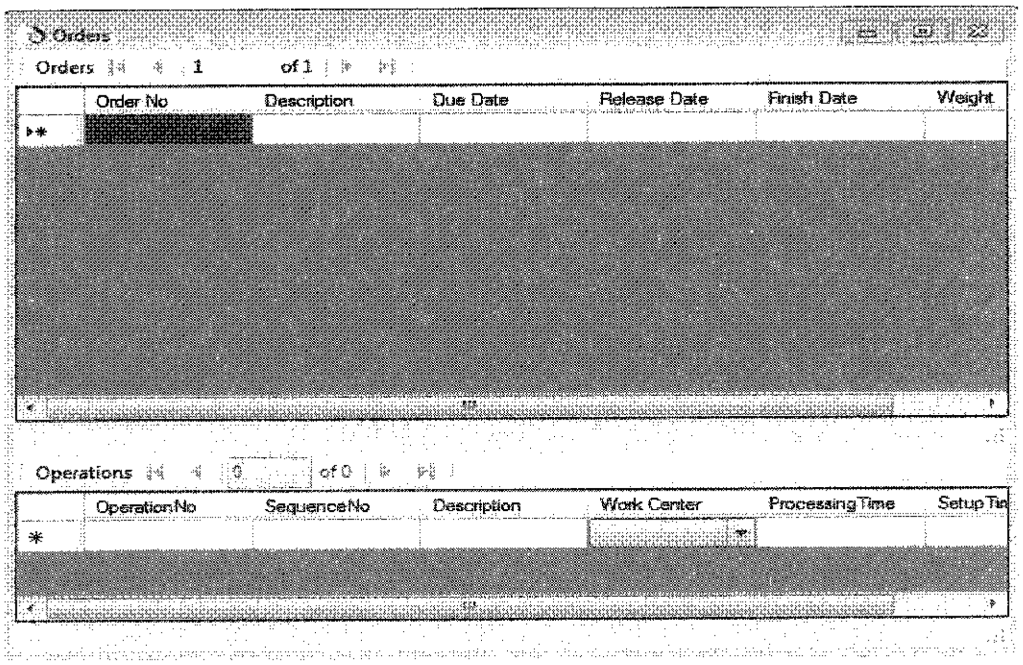


FIGURE 31. Order screen.

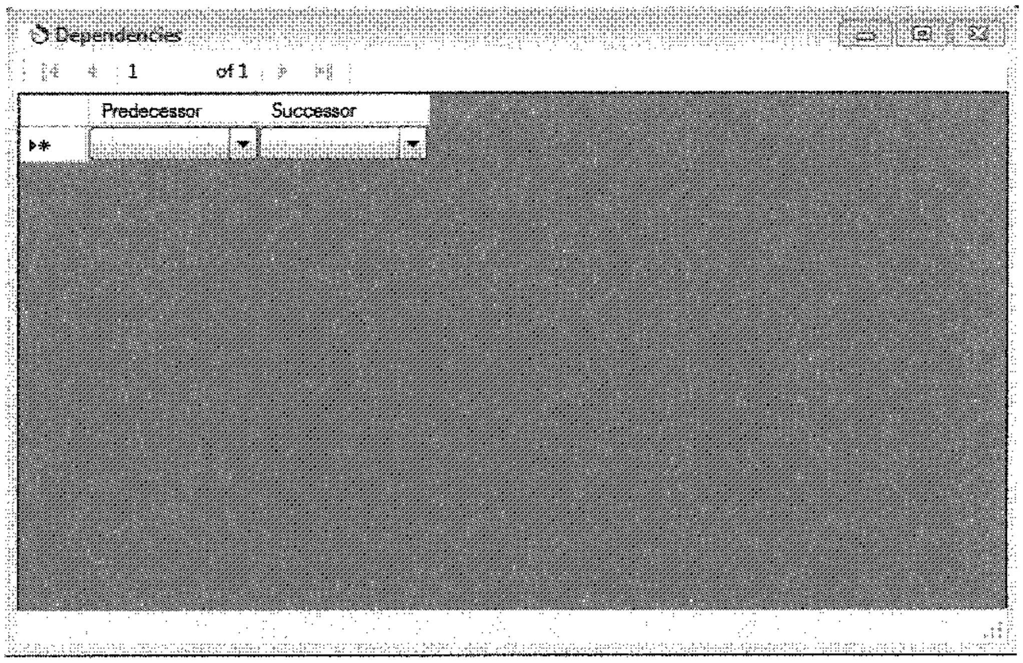


FIGURE 32. Dependencies screen.

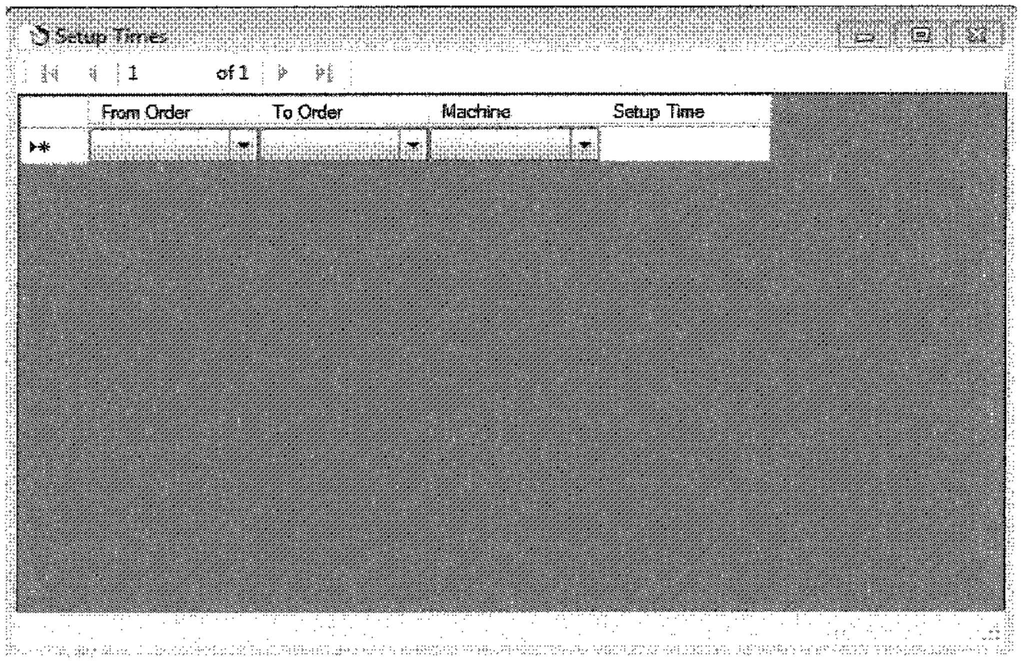
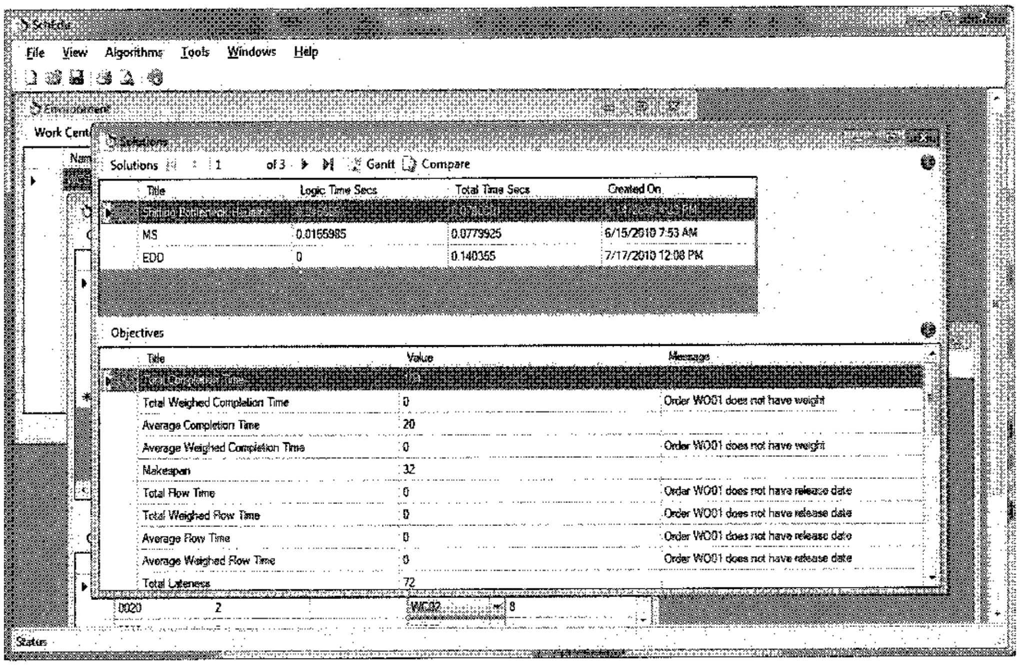


FIGURE 33. Setup time screen.

Figure 34 shows functionality added to the WinApp application to minimize the amount of work required to implement algorithms. The Solutions screen allows developers to run different versions of their algorithms, visualize the results using Gantt charts, view the results of their objective functions applied to each solution and also compare objective functions from different solutions.


SchEdu
File View Algorithms Tools Windows Help
Environment
Work Cent
Solutions
Solutions 1 of 3 Gantt Compare
Objectives
Status

Title	Logic Time Secs	Total Time Secs	Created On
Shifting Bottleneck Heuristic	[illegible]	[illegible]	[illegible]
MS	0.0155985	0.0779925	6/15/2010 7:53 AM
EDD	0	0.140355	7/17/2010 12:08 PM

Title	Value	Message
Total Completion Time	[illegible]	
Total Weighed Completion Time	0	Order WO01 does not have weight
Average Completion Time	20	
Average Weighed Completion Time	0	Order WO01 does not have weight
Makespan	32	
Total Flow Time	0	Order WO01 does not have release date
Total Weighed Flow Time	0	Order WO01 does not have release date
Average Flow Time	0	Order WO01 does not have release date
Average Weighed Flow Time	0	Order WO01 does not have release date
Total Lateness	72	

FIGURE 34. Solutions view.

Developers can easily compare the results of each of their algorithms by simply selecting all the solutions and using the "Compare" functionality. The Compare functionality displays a comparison chart for each objective functions.

CHAPTER 5

RESULTS AND DISCUSSION

The framework has been tested with the implementation of the following algorithms:

1. FAM-MS – First Available Machine Minimal Slack.
2. FAM-EDD – First Available Machine Earliest Due Date.
3. FAM-SPT – First Available Machine Shortest Processing Time.
4. FAM-LPT – First Available Machine Longest Processing Time.
5. FAM-WSPT – First Available Machine Weighed Shortest Processing Time.
6. Lawler.
7. Johnson.
8. Moore.
9. SBH - Shifting Bottleneck Heuristic.

The algorithms above have different properties. This selection was done in order to test the framework against problems with different characteristics.

FAM-MS, FAM-EDD, FAM-SPT, FAM-LPT and FAM-WSPT

These first four dispatching rules have very similar implementations using the scheduling framework. Therefore, the results have been analyzed together.

The logic for all of them is very similar it could be described by the steps below:

```
When machines are available
        Find next set of jobs that could be scheduled on the available machines
                Apply the desired dispatching rule
```

The minimum slack dispatching rule has been the first algorithm implemented. Once a machine is available, the minimum slack is applied to the list of possible orders that could be executed on the available machine. The rule consists in calculating the difference between the amount of processing time still required to finish the order and the due date of the order. The order with the minimum value for this difference is the one with minimum slack.

The whole implementation of the FAM-MS dispatching rule is listed below:

```
namespace FAM_MS
{
  class FAM_MSAlgorithm : FAM
  {
    protected override SchEduDataSet. OperationRow ApplyDispatchRule(int t, SchEduDataSet.
MachineRow machine, OperationList operations)
    {
      int minSlack = Int32. MaxValue;
      SchEduDataSet. OperationRow nextOperation = null;
      foreach (SchEduDataSet. OperationRow op in operations)
      {
        int slack = op. OrderRow. DueDate - OrderOperationSortedList[op. OrderRow].
GetProcessingTime();
        if (slack < minSlack)
        {
          nextOperation = op;
          minSlack = slack;
        }
      }

      return nextOperation;
    }
    public override AboutInfo AboutInfo
    {
      get
      {
        AboutInfo aboutInfo = new AboutInfo();
        aboutInfo. Title = "FAM MS";
        aboutInfo. Description = Resources. FAM_MSAlgorithmDescription;
        aboutInfo. Guid = new Guid("{C8AE55A1-A159-4af9-9F43-93BB381CC540}");
        return aboutInfo;
      }
    }
  }
}
```

The only relevant method in the FAM_MS class above is ApplyDispatchRule. The AboutInfo method is just self documenting mechanism available in the framework to allow developers to provide information such as algorithm's description and title to the users. It is reasonably safe to state the entire implementation could be represented by the fragment below.

```
protected override SchEduDataSet. OperationRow ApplyDispatchRule(int t, SchEduDataSet. MachineRow
machine, OperationList operations)
{
      int minSlack = Int32. MaxValue;
      SchEduDataSet. OperationRow nextOperation = null;
      foreach (SchEduDataSet. OperationRow op in operations)
      {
        int slack = op. OrderRow. DueDate - OrderOperationSortedList[op. OrderRow].
GetProcessingTime();
        if (slack < minSlack)
        {
          nextOperation = op;
          minSlack = slack;
        }
}
```

We observe that it took a total of 14 lines of code to implement the FAM-MS algorithm. The following services, which are typically needed of an equivalent development, have been provided by the framework:

1. Data entry services for inputting data.

2. Data storage and retrieval services for saving and loading the data.

3. Data validation for guarantying that the data is consistent to be used by the algorithm.

4. Data classes available in the DataModel namespace to manipulate orders and operations.

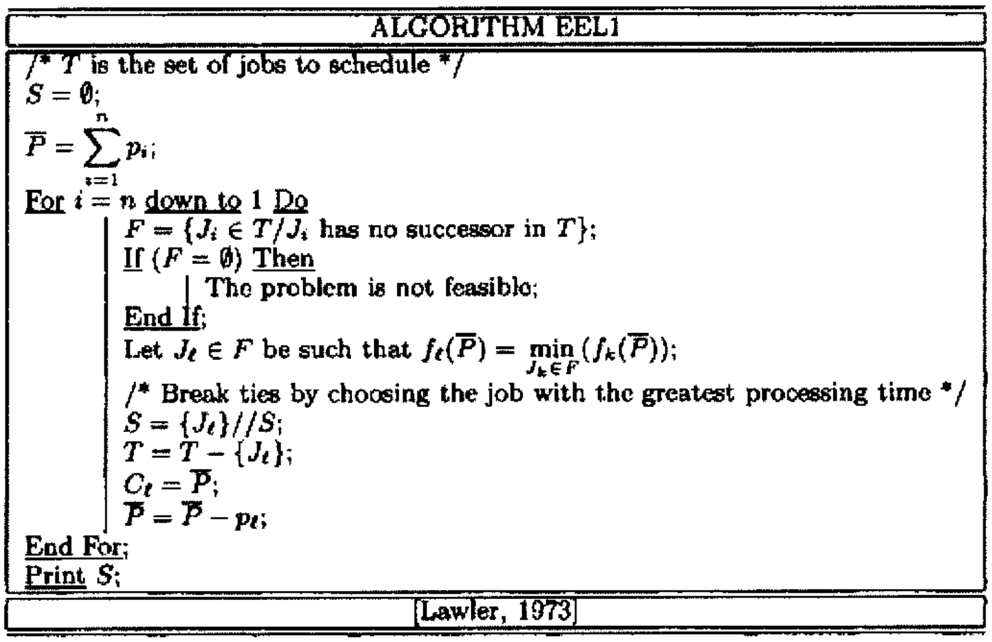
5. Presentation services to display the result to the user on a Gantt chart.

Although the FAM-MS has not been implemented without the framework in order to support analyzing the result, it is fair to expect that if the same algorithm was implemented without the help of the framework, a very significant number of lines of code would be required to achieve the same results.

It is important to notice that the FAM-MS has been implemented with the use of the FAM base class provided the framework. The same class has been used to implement FAM-EDD, FAM-LPT, FAM-SPT and FAM-WSPT with an average number of lines small than the FAM-MS implementation. The FAM class seems to provide a very simple mechanism to implement any desired dynamic dispatching rule. Moreover, the implementation did not require the creation of any new classes. The classes available in the framework were enough for the whole implementation.

Lawler's Algorithm for Minimizing Lateness and Tardiness

The second algorithm developed using the scheduling framework was the Lawler's algorithm for 1|prec| f_{max} . Two different cost functions have been used for f_{max}: lateness and tardiness. Figure 35, presented by T'Kindt (1), provides a pseudo algorithm for implementing the Lawler's algorithm.



```
ALGORITHM EEL1
/* $T$ is the set of jobs to schedule */
$S = \emptyset$;
$\overline{P} = \sum_{i=1}^{n} p_i$;
For $i = n$ down to 1 Do
    $F = \{J_i \in T / J_i$ has no successor in $T\}$;
    If ($F = \emptyset$) Then
        The problem is not feasible;
    End If;
    Let $J_\ell \in F$ be such that $f_\ell(\overline{P}) = \min_{J_k \in F}(f_k(\overline{P}))$;
    /* Break ties by choosing the job with the greatest processing time */
    $S = \{J_\ell\} // S$;
    $T = T - \{J_\ell\}$;
    $C_\ell = \overline{P}$;
    $\overline{P} = \overline{P} - p_\ell$;
End For;
Print $S$;
[Lawler, 1973]
```

FIGURE 35. Lawler's algorithm.

```
namespace Lawler
{
  public class LawlerLatenessAlgorithm : LawlerAlgorithm
  {
    public LawlerLatenessAlgorithm()
      : base()
    {
    }
    public LawlerLatenessAlgorithm(SchEduData schEduData)
      : base(schEduData)
    {
    }
    protected override int GetCost(SchEduDataSet. OrderRow order, int t)
    {
      return (t - order. DueDate);
    }

    public override AboutInfo AboutInfo
    {
      get
      {
        AboutInfo aboutInfo = new AboutInfo();
        aboutInfo. Title = "Lawler (Min Lateness)";
        aboutInfo. ObjectiveGUID = new Guid("{C1EC4D6C-6897-4e0b-B790-D50AEE44AE66}");
        return aboutInfo;
      }
    }

  }
```

The code above shows the implementation of the Lawler's algorithm to minimize lateness. The development was extremely simple due to the fact that the framework has provided a Lawler's base class. The implementation had only to derive from the Lawler's base class and provide the cost function. The only relevant code in the implementation of the Lawler's algorithm for minimizing lateness is in the GetCost function. The exactly same number of lines was needed for the implementation of Lawler's algorithm to minimize tardiness. The only difference between these 2 implementations was in the cost function. Therefore, both algorithms have been implemented with three lines.

Similarly to what happened during the implementation of the first four algorithms, the developer was able to implement the entire algorithm by providing the implementation of a single abstract method. Everything else has been provided by the framework.

Johnson's Algorithm

The Johnson's algorithm provides an optimal solution for minimizing makespan of a flow job problem with 2 machines. Orders have 2 operations. Operations must be processed on the first work center and then the second one.

The Johnson's logic schedules all operations for the first work center ordered ascending by processing time and then on the second work center ordered descending by processing time.

Similarly to the other algorithms implemented so far, the Johnson's algorithm was easily implemented by overriding two base class virtual methods: InitializeValidator and Logic. The list below shows the two relevant methods:

```
protected override void InitializeValidators()
{
 base. InitializeValidators();

 validators. Add(new WorkCenterValidator(SchEduData,2,2));
 validators. Add(new MachinePerWorkCenterValidator(SchEduData, 1, 1));
 validators. Add(new OperationWorkCenterValidator(SchEduData));
 validators. Add(new OperationPerOrderValidator(SchEduData, 2, 2));
 validators. Add(new OperationSequenceNoValidator(SchEduData));
 validators. Add(new FlowShopOperationValidator(SchEduData));

}

protected override void Logic()
{
    OperationList c1 = new OperationList();
    OperationList c2 = new OperationList();
    SchEduDataSet. WorkCenterRow workCenter1 = WorkCenters[0];
    SchEduDataSet. WorkCenterRow workCenter2 = WorkCenters[1];

    Operations. SortByProcessingTime();

    foreach (SchEduDataSet. OperationRow op in Operations)
    {
      if (op. WorkCenterID == workCenter1. ID)
      {
        c1. Add(op);
      }
      else
      {
        c2. Insert(0, op);
      }
    }

    OperationList c = new OperationList();
    c. AddRange(c1);
    c. AddRange(c2);

    // set start/end dates
    SetOperationDates();
  }
```

The total number of lines used to implement the Johnson's algorithm was around 30. Unlike Lawler's implementation, the framework did not provide any specialized class. That did not represent an issue for the implementation. The whole algorithm was developed with classes provided by the framework. It is important to notice that this time the implementation was required to override the InitializeValidator to guarantee that the data is in accordance with the Johnson's problem. The implementation was able to make use of validators available in the framework and there was no need to implement new validators.

Moore's Algorithm

Moore's algorithm solves the problem 1|d$_i$|Ū. A set of jobs must be scheduled on a single machine. Jobs have due date and preemption is not allowed. The objective function to be minimized is the number of late jobs. A pseudo algorithm, presented by T'Kindt (1), for Moore is depicted on Figure 36.

```
ALGORITHM EJM1
/* $T$ is the set of jobs to schedule */
/* We assume that $d_1 \leq d_2 \leq ... \leq d_n$ */
$S = (J_1, J_2, ..., J_n)$;
$Tardy = \emptyset$;
While ($\exists J_\ell \in S$ such that $C_\ell > d_\ell$) Do
    Let $k$ be such that $C_{S[k]} > d_{S[k]}$ and $\forall i < k,\ C_{S[i]} \leq d_{S[i]}$;
    Let $j$ be such that $j \leq k$ and $p_{S[j]} = \max_{i=1,...,k}(p_{S[i]})$;
    $S = S - \{J_j\}$;
    $Tardy = Tardy//\{J_j\}$;
End While;
$\overline{U} = |Tardy|$;
Print $S//Tardy$ and $\overline{U}$;
[Moore, 1968]
```

FIGURE 36. Moore's pseudo algorithm.

```
protected override void InitializeValidators()
{
  base. InitializeValidators();

  validators. Add(new OrderDueDateValidator(SchEduData));
  validators. Add(new WorkCenterValidator(SchEduData, 1, 1));
  validators. Add(new OperationPerOrderValidator(SchEduData, 1, 1));
}
```

```
  protected override void Logic()
  {
    OperationList partialScheduledOperations = new OperationList();
    OperationList operationWithGreatestProcessinTime = new OperationList();

    SchEduDataSet. WorkCenterRow workCenter = SchEduData. WorkCenter[0];
    SchEduDataSet. MachineRow machine = SchEduData. Machine[0];

    SchEduDataSet. OperationRow maxProcessingTime;

    // Sort operations by EDD
    Operations. SortByEDD();

    // set start date, end date and assign the work center and machine to operations
    SetOperationDates();

    foreach (SchEduDataSet. OperationRow op in Operations)
    {
      // Add operation to the partial scheduled list
      partialScheduledOperations. Add(op);

      // check if operation is late
      if (op. EndDate > op. OrderRow. DueDate)
      {
        continue;
      }
      // Operation is late, remove the operation with the greatest processing time and move to
operationWithGreatestProcessinTime list
      maxProcessingTime = partialScheduledOperations. MaxProcessingTime();
      partialScheduledOperations. Remove(maxProcessingTime);
      operationWithGreatestProcessinTime. Add(maxProcessingTime);
    }

    Operations. Clear();
    Operations. AddRange(partialScheduledOperations);
    Operations. AddRange(operationWithGreatestProcessinTime);

    // recalculate their start dates and end dates
    SetOperationDates();
    }
```

The implementation of Moore's algorithm has been accomplished by implementing only two methods: InitializeValidators and Logic. Similarly to what happened to the Johnson's implementation.

The InitializeValidator method has allowed the implementation to provide validations that are required to guarantee the compliance of the data with the Moore 'sproblem. The developer did not have to implement any new validators, because the framework had all the validators required. The developer had only to add the desired validators to the stack of validators to be invoked before the execution of the logic.

The total number of required by the Moore's implementation was 32 (excluding the documentation code and comments). The developer again was not required to create any new classes.

SBH–Shifting Bottleneck Heuristic Algorithm

The SBH was the last and the most complex algorithm implemented on this study. The shifting bottleneck heuristic is one of the most successful heuristic procedures developed for $Jm||C_{max}$ (2). The basic idea of the SBH is to find the machine with maximum lateness (bottleneck) and the try to find the sequence of operations that minimized the lateness on this machine. The step to find the sequence that minimized the lateness on the selected machine is a known NP-hard defined by $1|r_j|L_{max}$. There are different techniques, such as branch and bound, that could be used to improve the algorithms execution time. Since the scope of this work is to analyze the benefits of using a framework, the implementation for the SBH did not use any special technique and it is just analyzing all the possible permutations when looking for the solution of

$1|r_j|L_{max}$. The code below shows the implementation of the SBH using the scheduling framework.

```
protected override void InitializeValidators()
{
  base. InitializeValidators();

  validators. Add(new OperationWorkCenterValidator(SchEduData));
  validators. Add(new MachinePerWorkCenterValidator(SchEduData,1,1));
  validators. Add(new OperationPerOrderValidator(SchEduData, 1, -1));
  validators. Add(new OperationSequenceNoValidator(SchEduData));

}

int Cmax = Int32. MinValue;
WorkCenterOperationList workCenterOperationSequence = new WorkCenterOperationList();

protected override void Logic()
{

  Cmax = Int32. MinValue;
  workCenterOperationSequence = new WorkCenterOperationList();

  // Calculate longest path
  Cmax = OrderOperationSortedList. GetCmax();

  while (WorkCenterOperationList. Count > 0)
  {

    SetReleaseDates();
    SetDueDates();

    // find bottleneck
    KeyValuePair<WorkCenterKey, OperationList> workCenterBottleneck = new
KeyValuePair<WorkCenterKey, OperationList>();
    int bottleneckLmax = Int32. MinValue;
    foreach (KeyValuePair<WorkCenterKey, OperationList> a in WorkCenterOperationList)
    {
      OperationList minLmaxOperationList = MinimizeLmax(a. Value);
```

```
          int Lmax = CalculateLateness(minLmaxOperationList);
          if (Lmax > bottleneckLmax)
          {
            workCenterBottleneck = new KeyValuePair<WorkCenterKey, OperationList>(a. Key,
minLmaxOperationList);
            bottleneckLmax = Lmax;
          }
        }
        CalculateLateness(workCenterBottleneck. Value);
        workCenterOperationSequence. Add(workCenterBottleneck. Key, workCenterBottleneck. Value);
        WorkCenterOperationList. Remove(workCenterBottleneck. Key);
        Cmax += bottleneckLmax;
      }

      CreateAllocation();
    }

    protected OperationList MinimizeLmax(OperationList operations)
    {
      OperationList op = new OperationList();
      IEnumerable<IEnumerable<SchEduDataSet. OperationRow>> permutations = Permuter.
Permute(operations);
      int minLmax = Int32. MaxValue;
      foreach (IEnumerable<SchEduDataSet. OperationRow> p in permutations)
      {
        int lateness = CalculateLateness(p);
        if (lateness < minLmax)
        {
          op =  new OperationList(p);
          minLmax = lateness;
        }
      }
      return op;
    }

    protected int CalculateLateness(IEnumerable<SchEduDataSet. OperationRow> operations)
    {
      int availabilityDate = 0;
      int Lmax = Int32. MinValue;
      foreach(SchEduDataSet. OperationRow op in operations)
    {
      if (availabilityDate < op. AvailabilityDate)
      {
        availabilityDate = op. AvailabilityDate;
      }
      op. StartDate = availabilityDate;
      op. EndDate = availabilityDate + op. ProcessingTime - 1;
      int lateness = Math. Max(0, op. EndDate - op. DueDate);

      Lmax = Math. Max(Lmax, lateness);
      availabilityDate = op. EndDate + 1;
    }

    return Lmax;
  }

  protected void SetReleaseDates()
  {
    foreach (KeyValuePair<SchEduDataSet. OrderRow, OperationSortedList> a in
OrderOperationSortedList)
    {
      int availabilityDate = 0;
      foreach (SchEduDataSet. OperationRow op in a. Value. Values)
      {
        if (workCenterOperationSequence. ContainsWorkCenter(op. WorkCenterRow) == true)
        {
          availabilityDate = op. EndDate + 1;
          continue;
        }
        op. AvailabilityDate = availabilityDate;
        availabilityDate += op. ProcessingTime;
      }
    }
  }

  protected void SetDueDates()
  {
    foreach (KeyValuePair<SchEduDataSet. OrderRow, OperationSortedList> a in
OrderOperationSortedList)
    {
        int dueDate = 0;
        for (int i = a. Value. Count - 1; i >= 0; i--)
        {
          if (workCenterOperationSequence. ContainsWorkCenter(a. Value. Values[i]. WorkCenterRow) ==
true)
          {
            continue;
          }

          a. Value. Values[i]. DueDate = Cmax - dueDate;
          dueDate += a. Value. Values[i]. ProcessingTime;
        }
      }
    }

    protected void CreateAllocation()
    {
      foreach(SchEduDataSet. WorkCenterRow workCenter in WorkCenters)
      {
        workCenter. AvailabilityDate = 0;
      }
      foreach(SchEduDataSet. OrderRow order in Orders)
      {
        order. AvailabilityDate = 0;
      }

      OperationList nextOperationByWorkCenter = workCenterOperationSequence. GetNextOperations();
      while (nextOperationByWorkCenter. Count > 0)
      {
        OperationList nextOperationByOrder = OrderOperationSortedList. GetNextOperations();
        SchEduDataSet. OperationRow nextOperation = null;
        foreach (SchEduDataSet. OperationRow op in nextOperationByWorkCenter)
        {
          if (nextOperationByOrder. Contains(op) == true)
          {
            nextOperation = op;
            break;
          }
        }
        if (nextOperation == null)
        {
          throw new SchEduException(Resource. NoSolutionMessage);
        }
        nextOperation. StartDate = Math. Max(nextOperation. WorkCenterRow. AvailabilityDate,
nextOperation. OrderRow. AvailabilityDate);
        nextOperation. EndDate = nextOperation. StartDate + nextOperation. ProcessingTime - 1;
        nextOperation. WorkCenterRow. AvailabilityDate = nextOperation. EndDate + 1;
        nextOperation. OrderRow. AvailabilityDate = nextOperation. EndDate + 1;

        workCenterOperationSequence. RemoveByOperation(nextOperation);
        OrderOperationSortedList. RemoveByOperation(nextOperation);
        nextOperationByWorkCenter = workCenterOperationSequence. GetNextOperations();
      }

      // Assign Machine
      foreach (SchEduDataSet. OperationRow operation in SchEduData. Operation)
      {
        operation. MachineID = operation. WorkCenterRow. GetMachineRows()[0]. ID;
      }

    }
```

Even though the number of lines of code required, around 100, for the SBH algorithm was significant larger than the other algorithms, it can be considered still a small number when the complexity of the algorithm is taken into consideration. Moreover, this number of lines when compared to any software project is still extremely small.

The SBH, like in the previous implementations, did not require the development of any other code but the logic of the algorithm itself. The framework has provided all the necessary screens, data classes, validators, and et cetera.

CHAPTER 6

CONCLUSIONS

After implementing the first two algorithms, FAM-MS and Lawler's, it was possible to notice a significant straightforwardness on how the development of both algorithms was done. However, to state that the framework can support a broad range of different problems and always reduce the amount of work required for implementing any scheduling algorithm was too early. Both first implementations were extremely simple due to the fact that the framework has provided specialized base classes for a set of problems that fall either into Lawler's or dynamic dispatching category of problems.

The three other algorithms were, however, implemented without the support of any specialized classes. The whole development was done just with general classes present in the framework. A set of commonalities were found in these implementations, which is a good indication that it is possible to generalize scheduling problems and have more functionality at framework level to be shared by different algorithms.

The use of the framework during the development has allowed the developer to focus on the implementation of the algorithm's logic itself. There was no need to deal with any other tasks such as reading data, saving data, or user interfaces. The framework has provided all the foundation required for all implemented algorithms.

There was not need to create any extra data types such as list, collections, or arrays. The framework has provided all the data types that were required for the implementation of all algorithms on this study. The framework has also provided all the user interfaces required by the development. Data entry and results visualizations were available and the developer did not have to develop any extra code to use it.

Finally, the number of lines required was very small. With the exception of FAM-MS, four dispatching rules, FAM-EDD, FAM-SPT, FAM-LPT and FAMWSPT were all implemented in one single line. Algorithms such as Johnson's or Moore have been implemented with about 30 lines and complex algorithms such as the Shifting Bottleneck Heuristic with about 100. These numbers are extremely small when compared to any metric for lines of code in software development.

CHAPTER 7

FUTURE DEVELOPMENTS

The abstractions and base classes created in the current development of the framework were defined based on the analysis and implementation of a few scheduling algorithms. Due to time constraint, it was not possible to test the framework with a broad and more heterogeneous collection of scheduling problems. In the next development iteration performed on the framework, a broader set of algorithms should be implemented to validate the framework usability and to abstract more similarities among algorithms.

The current implementation has not added any support for non-reusable resources (sometimes called disposable resources). Although reusable resources are the most common type of resource dealt with by scheduling applications, the presence of non-reusable cannot be disregarded completely. It is quite common to find resources such as drill bits that, when used a certain number of times, must be disposed. Therefore, a mechanism for handling different types of resources is an important improvement to be considered.

Support to calendars is another improvement necessary to develop the framework into in a full fledge framework. Most companies have well-defined working shifts. Working shifts inform when resources are available to be allocated. Some factories might work from 8:00 a.m. to 5:00 p.m., five days a week, while others might work 24 by

7. Scheduling applications must take into consideration the availability of all resources while creating any schedule.

REFERENCES

REFERENCES

(1) Pinedo, Michael L. Scheduling Theory, Algorithms and Systems. New York: Springer, 2008.

(2) Rajpathak, Dnyanesh, Motta, Enrico and Roy, Rajkumar. A Generic Library of Problem Solving Methods for Scheduling Applications. IEEE Transactions on Knowledge and Data Engineering. 2006, pp. 815-828.

(3) Smith, Stephen F and Becker, Marcel A. An Ontology for Constructing Scheduling System.

(4) Graham, R. L.,E. L. Lawler,J. K. Lenstra,A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling theory: A survey. Annals of Discrete Mathematics. 1979, pp. 287-326.

(5) J. Blazewick, K. Ecker, E. Pesch, G. Schmidt, J. Weglarz. Handbook on Scheduling: From Theory to Applications. s.l.: Springer, 2007.

(6) Oyetunji, E.O. Some Common Performance Measures in Scheduling Problems: Review Article. Research Journal of Applied Sciences, Engineering and Technology 1(2): 6-9, 2009.

(7) T'Kindt, Vincent and Billaut, Jean Charles. Multicriteria Scheduling: Theory, Models and Algorithms. Tours, France: Springer, 2005.

(8) Bruker, Peter. Scheduling Algorithms. Osnabrück, Germany: Springer, 2006.

(9) A Generic Task Ontology for Scheduling Applications. D. Rajpathak, E. Motta, R. Roy. Nevada, Las Vegas, USA: s.n., 2001. International Conference on Artificial Intelligence.

(10) Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. Roberts, Don and Johnson, Ralph. s.l.: Addison-Wesley, 1996. Proceedings of the Third Conference on Pattern Languages and Programming.

(11) Gamma, Erich, et al. Design Patterns: Elements of Reusable Object-Oriented Software. s.l.: Addison-Wesley, 1997.

(12) Lawler, E. L. Optimal sequencing of a single machine subject. Management Science. 1973, pp. 544-546.

(13) Cwalina, Krzysztof and Abrams, Brad. Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries. s.l.: Microsoft.

(14) Fowler, Martin and Scott, Kendall. UML Distilled: A Brief Guide to the Standard Object Modeling Language (2nd Edition). s.l.: Addison-Wesley, 1999.

(15) Markiewicz, Marcus Eduardo and Lucena, J.P. Carlos. Object Oriented Framework Development. The ACM Student Magazine.