

DEVELOPMENT OF AN OBJECT ORIENTED MATHEMATICAL MODELING
SYSTEM

by

Tuğba Atmaca

B.S., Management Information Systems, Boğaziçi University, 2008

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Industrial Engineering
Boğaziçi University

2011

ACKNOWLEDGEMENTS

First and foremost I offer my sincerest gratitude to my supervisors, Assoc. Prof. Ali Tamer Ünal and Assist. Prof. Zeki Caner Taşkın, for their wisdom and endless contributions throughout this thesis. They were always understanding, motivating, and patient. I could not have imagined having better and friendlier advisors. Apart from this thesis, they have always been the ones whose suggestions I truly needed.

I would like to thank Assoc. Prof. Aşlı Sencer Erdem firstly for teaching industrial engineering related courses in a so kind and informative manner that my interest in that field increased noticeably and leaded me to study in an Industrial Engineering master program. I am also grateful for her guidance, support and taking part in my thesis jury.

I am grateful to all my friends for supporting me spiritually but especially Zehra Melis Teksan for her friendship and help during my hardest times.

I am also grateful to my family for their continuous support and unconditional love for me throughout this thesis and my whole life.

Lastly, I gratefully acknowledge the support of TUBITAK - Turkish Technological and Scientific Research Institute during my thesis period.

This research has been partly supported by the Boğaziçi University Research Fund Grant No. 5028.

ABSTRACT

DEVELOPMENT OF AN OBJECT ORIENTED MATHEMATICAL MODELING SYSTEM

In this study, ICRON, which is a tool supporting object oriented mathematical programming, is partially developed in order to reduce the workload during the modeling and solving process of large scale optimization problems. The effectiveness of decomposition techniques in solving large scale optimization problems have led to the development of several modeling environments and tools having their own functionalities in order to increase the performances. These environments and tools in the literature have both some advantages and some disadvantages with respect to the support of easy modeling and fast solving. The new proposed mathematical programming tool is developed to allow better modeling environment to the user and to get shorter solution time relative to the previous version of ICRON by eliminating the repetitions both on models and on solver iterations which are of a considerable number especially when implementing a decomposition technique. One of the important aspects of the tool is its object oriented architecture that provides object oriented modeling of mathematical programming problems by using real world object descriptions instead of writing codes depending on sets and indexes. More importantly, it allows the user to construct a structure keeping the whole model and solver information throughout running of algorithms which provides flexibility to the user to define objective functions, decision variables or constraints and, to update the current solver information via special constructs to add column, to add row, to relax constraints, and to change types of decision variables in anywhere of the algorithm. The strong relationship between business objects and mathematical programming objects in the environment is due to the tight integration of the modeler and the solver which brings the tool beyond of the others.

ÖZET

NESNE TABANLI BİR MATEMATİKSEL MODELLEME SİSTEMİNİN GELİŞTİRİLMESİ

Bu çalışmada, büyük ölçekli eniyileme problemlerinin modellenmesi ve çözülmesi süresince gerçekleşen iş yükünü azaltmak üzere nesne tabanlı matematiksel programlamayı destekleyen bir yazılım olan ICRON kısmen geliştirilmiştir. Büyük ölçekli problemlerin çözümünde ayrıştırma tekniklerinin etkinliği, kendi performanslarını artırmak üzere geliştirilmiş fonksiyonları olan çeşitli modelleme ortamları ve araçlarının gelişmesine yol açmıştır. Literatürdeki bu ortam ve araçların kolay modelleme desteği ve hızlı çözme açısından hem bazı artıları hem de bazı eksileri vardır. Yeni önerilen matematiksel programlama aracı, kullanıcıya daha iyi modelleme ortamı sağlamak için ve özellikle ayrıştırma tekniklerini uygularken hem modellerde hem de çözücüde çok sayıda olan tekrarlamaları ortadan kaldırmak suretiyle ICRON'un önceki sürümüne kıyasla daha kısa çözüm süresi elde etmek için geliştirilmiştir. Aracın önemli yönlerinden biri, matematiksel programlama problemlerini setlere ve endekslere dayanan kodların yazımı şeklinde değil de, gerçek dünya nesnelerinin tanımlamaları ile gerçekleştirmeyi sağlamayan nesne tabanlı bir mimariye sahip olmasıdır. Daha da önemlisi, bu araç kullanıcıya, algoritmalar çalışırken tüm model ve çözücü bilgisini akılda tutan bir yapı inşa etmesine imkan verir ki bu, kullanıcıya algoritmanın herhangi bir yerinde amaç fonksiyonu, karar değişkeni ya da kısıtlama tanımlamak ya da özel olarak sütun eklemek, satır eklemek, kısıtlamaları gevşetmek ve karar değişkenlerinin tipini değiştirmek için geliştirilen yapıları kullanarak mevcut çözücü bilgisi üzerinde güncellemek yapabilmek için kullanıcıya esneklik sağlar. İş nesneleri ve matematiksel programlama nesnelerinin ortamdaki güçlü ilişkisi modelleyici ve çözücü arasındaki sıkı bütünleşmesinden kaynaklanmaktadır ki bu da aracı diğerlerinin ötesine taşımaktadır.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	iv
ÖZET	v
LIST OF FIGURES	viii
LIST OF TABLES.	xi
LIST OF SYMBOLS	xii
LIST OF ACRONYMS/ABBREVIATIONS	xiii
1. INTRODUCTION	1
2. LITERATURE REVIEW	6
2.1. Large Scale Optimization and Decomposition	6
2.2. Frameworks and Scripting Languages	8
2.3. Algebraic Systems	9
2.3.1. GAMS	9
2.3.2. AMPL	10
2.3.3. LINDO/LINGO	11
2.3.4. MPL	11
2.3.5. AIMMS	11
2.4. Object Oriented Systems	12
2.5. Modeling Tools Integrated into Programming Languages	15
2.5.1. FLOPC++	15
2.5.2. ILOG Concert Technology	15
2.5.3. Java ILP	16
2.5.4. OptimJ	16
2.5.5. Xpress-Mosel / Xpress-BCL	17
2.5.6. PuLP and Dippy	18
3. PROPOSED MODELING ENVIRONMENT	19
3.1. Proposed Object Model	25
3.2. Proposed Mathematical Modeling Nodes	27
3.2.1. Create Model	27
3.2.2. Create Model Objective.	29

3.2.3. Add Model Objective	32
3.2.4. Set Model Decision Variable	34
3.2.5. Create Model Constraint	36
3.2.6. Model Solve	40
3.2.7. Apply Model Solution	43
3.2.8. Create Column	44
3.2.9. Add Column	46
4. CASE STUDIES USING ICRON	49
4.1. An Introductory Example: Cutting Stock Problem	49
4.1.1. The Definition of a Cutting Stock Problem	49
4.1.2. An Example of Cutting Stock Problem	50
4.1.3. Modeling with ICRON	53
4.1.4. Solution of Cutting Stock Problem	57
4.2. Cutting Stock Problem Using Gilmore-Gomory Column Generation Algorithm	60
4.2.1. Column Generation Algorithm	60
4.2.2. Gilmore-Gomory Algorithm	62
4.2.3. Modeling with ICRON	63
4.2.4. Solution of Cutting Stock Problem	81
4.2.5. Adding a New Constraint to the Cutting Stock Problem	85
5. CONCLUSION	87
APPENDIX	89
REFERENCES	97

LIST OF FIGURES

Figure 1.1. Model-solver communication tightness.	4
Figure 3.1. Communications between main entities of MP in ICRON.	20
Figure 3.2. ICRON-Solver communication.	22
Figure 3.3. Class diagram of proposed system.	26
Figure 3.4. Create model node.	28
Figure 3.5. Dialog box of create model node.	28
Figure 3.6. Running of create model node.	29
Figure 3.7. Create model objective node.	30
Figure 3.8. Dialog box of create model objective node.	30
Figure 3.9. A defined create model objective node.	30
Figure 3.10. Running of create model objective node.	32
Figure 3.11. Add model objective node.	32
Figure 3.12. Dialog box of add model objective node.	33
Figure 3.13. Running of add model objective node.	33
Figure 3.14. Set model decision variable node.	34
Figure 3.15. Dialog box of set model decision variable node.	35

Figure 3.16. Running of set model decision variable node.	36
Figure 3.17. Create model constraint node.	37
Figure 3.18. Dialog box of create model constraint node.	38
Figure 3.19. Create model constraint node for a demand constraint.	38
Figure 3.20. A demand constraint example.	38
Figure 3.21. Running of create model constraint node.	40
Figure 3.22. Model solve node.	40
Figure 3.23. Running of solve model node.	42
Figure 3.24. Apply model solution node.	43
Figure 3.25. Running of apply model solution node.. . . .	43
Figure 3.26. Create column node.	44
Figure 3.27. Running of create column node.	45
Figure 3.28. Add column node.	46
Figure 3.29. Dialog box of add column node.	47
Figure 3.30. Running of add column node.	48
Figure 4.1. Different patterns on a roll.	50
Figure 4.2. Class diagram of a cutting stock problem.	53
Figure 4.3. Data model of cutting stock problem in ICRON.. . . .	54

Figure 4.4. Mathematical model of cutting stock problem in ICRON.	55
Figure 4.5. Communication between master problem and sub-problems.. . . .	61
Figure 4.6. The initial set.. . . .	62
Figure 4.7. The class diagram of the cutting stock using Gilmore-Gomory Algorithm	64
Figure 4.8. Data model.. . . .	65
Figure 4.9. Mathematical model of cutting stock problem using the Gilmore-Gomory Algorithm.	67
Figure 4.10. Add Column node in the CuttingStock_Algorithm.	69
Figure 4.11. Master model.	73
Figure 4.12. The relation between Pattern and MP_Column objects and between PatternDetail and MP_Coefficient objects.	75
Figure 4.13. SubModel1 algorithm.	77
Figure 4.14. SubModel2 algorithm.	79
Figure 4.15. The cutting stock algorithm with additional constraints.	84

LIST OF TABLES

Table 4.1. Demand data	51
Table 4.2. All possible patterns.	51
Table 4.3. Mathematical model of cutting stock problem	58
Table 4.4. Optimal solutions of the cutting stock problem.	59
Table 4.5. The Initial set of the cutting stock example	71
Table 4.6. The mathematical model of submodel in the first call	81
Table 4.7. Optimal values of the submodel in the first call	81
Table 4.8. The mathematical model of submodel in the second call	82
Table 4.9. Optimal values of the submodel in the second call	82
Table 4.10. The pattern list in the last master model	82
Table 4.11. Mathematical model of cutting stock model using column generation .	83
Table 4.12. Optimal solutions of cutting stock model using column generation . .	83
Table 4.13. Optimal values of the cutting stock problem with new added constraints	86

LIST OF SYMBOLS

X_j	The Number of Times That Pattern j Will Be Used
A_{ij}	The Number of Product of Length i after Applying Pattern j
D_i	The Demand for Product with Length i
c_i	The Cost of Pattern i
π_i	The Dual of Demand Constraint Related With Product with Length i
A_i	The Number of Times Item i is Selected in The New Pattern
W_i	The Length of Product i
W	The Length of a Roll of Stock

LIST OF ACRONYMS/ABBREVIATIONS

AIMMS	Advanced Integrated Multidimensional Modeling Software
AML	Algebraic Modeling Language
AMPL	A Mathematical Programming Language
API	Application Programming Interfaces
AREMOS	A Refinery Modeling System
BARON	Branch and Reduce Optimization Navigator
BC-OPT	Branch-and-Cut Code for Mixed Integer Programs
BLOOMS	Basic Language Object Oriented for Modeling System
COIN	Computational Infrastructure for Operations Research
CONOPT	Constrained Optimization
DICOPT	Discrete and Continuous Optimizer
DIP	Decomposition for Integer Programming
EMOSL	Entity Modeling and Optimization Subroutine Library
FLOPC++	Formulation of Linear Optimization Problems in C++
GAMS	General Algebraic Modeling System
GSAMS	Graphical Scheduling Algorithm Modeling System
IBM	International Business Machines
LINDO	Linear Interactive an Discrete Optimizer
MGG	Matrix Generator Generator
MILP	Mixed Integer Linear Programming
MINTO	Mixed Integer Optimizer
MIPO	Multistage Inventory Planning and Optimization
MME	Mathematical Modeling Environment
MP	Mathematical Programming
MPS	Mathematical Programming System
ODBMS	Object Oriented Database Management System
OMAC	Operations Modeling and Analysis Core
OPL	Optimization Programming Language
OptimJ	Optimization in Java
SML	Structured Modeling Language

SNOPT	Sparse Nonlinear Optimizer
TechTool	Technical Tool
UML	Unified Modeling Language

1. INTRODUCTION

It is significant to have an adequate and well-defined representation of real life problem in the mathematical world. The better this projection is defined, the better the solution obtained from mathematical problem will represent the solution of the real problem. Thus, a good optimization system should ease the modeling efforts as much as possible so that adequate and reliable mathematical models can be constructed by the users of the system. Additionally, it should support its users by partially automating the representation of the real problem in modeling world and the back translation of solutions obtained from the optimization systems in terms of real objects (Sungur, 2003).

Since it is hard to deal with large-scale optimization problems, which usually better represent the real life problem, not only good optimization systems but also some techniques have been developed so as to reach the solution as soon as possible by controlling the factors which make the problem harder. Decomposition techniques in optimization theory have been developed for that reason, and it is based on the thought of the decomposition of a model into smaller parts instead of constructing a single complicated model, and progression on the solution at each iteration. One technique, which is generally used to make it possible to solve problems that would otherwise be too large or too difficult to solve, is Dantzig-Wolfe decomposition technique (Dantzig and Wolfe, 1960). This technique, which is appropriate for the problems with complicating constraints, uses column generation to create proposed solutions for the set of easy constraints. The basic process is as follows: a master problem containing at least the currently active columns (the basis) uses a sub-problem or sub-problems to generate columns for entry into the basis such that their inclusion improves the objective function. This technique relies on the fact that generating columns can be done much more quickly than solving the original problem.

As can be contrasted to Dantzig-Wolfe decomposition, which is based on column generation, Bender's decomposition (Benders, 1962), another main technique, is referred to as a row generation approach. The name comes from that this algorithm adds new constraints as it progress towards the solution. This technique also allows the solution of

very large linear programming problems provided that they have a special block structure with complicating variables.

Besides that decomposition techniques are used to decrease the time needed to get the solution of a large-scale model, the integer programming solvers, which are used to solve the mathematical problems with its own components and routines, have been developed so as to allow users to handle the parameter settings in order to change the behavior and performance of the optimizers to get the desired performance (Atamtürk and Savelsbergh, 2005). The state-of-the-art integer programming solvers such as CPLEX (ILOG, 2008), LINDO (LINDO Systems, 1999), and Xpress-MP (FICO XPress Optimization Suite 7.1, 2011) provide different availability to the users to modify the parameters to control these solvers' components to have a customized solution approach. Due to this type of customization opportunity, it is possible to implement dynamic cut, column generation, branching, or heuristics to get feasible solutions more rapidly. However, these developments load a heavy burden to users by requiring them to learn different software and different libraries whenever they want to use a different solver.

In addition to the difficulty in learning each solver structure, the considerable programming effort required to prepare the data into a format recognizable by the solver is also cumbersome. This format typically requires creating a constraint matrix with coefficients and associated row and column indices. As a result of considering the difficulty of creating such a representation and making modifications on it, there have been many research studies to ease this process. Beginning with the definition of Mathematical Programming System (MPS) format by IBM in early sixties, and continuing with the development of matrix generation languages such as DATAFORM, OMNI, and MGG, the studies to have a standard for defining and manipulating MP problems increased with the technology. After the matrix generation languages, the algebraic modeling languages like GAMS, AMPL, AIMMS, and LINGO became the basis of the next generation of modeling languages. Such developments in the modeling languages have made it possible to model more large-scale problems. Since mixed integer programming models better represent the real business problems, several frameworks were developed in 1990s to solve the problems by providing the user the ability to add their own problem specific routines for customizing both the branching and the bounding operations; for example, MINTO (Nemhauser et al.,

1994), MIPO (Balas et al., 1996), BC-OPT (Cordier et al., 1999), SIP (Martin, 1998), ABACUS (Jünger and Thienel, 1998), and SYMPHONY (Ralphs and Güzelsoy, 2005). By considering the great flexibility provided by the most of the software frameworks to the users to implement any column generation method which pertains to their specific application, DIP (Galati, 2010), (DIP, 2010) has been developed to tie together various algorithmic approaches related to decomposition methods in order to reduce the burden of users emanated from this great flexibility.

Although modification of parameter settings can be a good way to solve many integer programs, there remain cases in which this is not sufficient. In such cases, it may be necessary to develop decomposition or iterative approaches. In order to easily develop and experiment with such approaches, a close integration of modeling tools and optimization engines is required. As shown in Figure 1.1, the level of tightness of the integration between modeler and the solver is demonstrated by considering the performances of mathematical programming tools especially during calling the solver several times. Some of the algebraic systems more tend to be at the lower levels due to their modeling limitations and complexities. Relative to the other tools GAMS forces the user to construct a model to call the solver several times. With sets, indices, and routines, a model is built in borders and then run as a block. Thus, the solver is expected to take the whole data of this block in each call. Here, it is a shortcoming of the system that a change in the model does not have its direct representation in the solver immediately. Hence, the system is lower levels in terms of integration tightness. Then, a better system is also an algebraic modeling tool; AMPL. It has commands that will manage the relationship between the data, model and the solver such as the selection of solver, determining the model and data files, defining relax and presolve options, creating and controlling loops between models, and using the solution output. Another algebraic modeling system, AIMMS, has more opportunities about modeling and calling solver. By using this system, it is possible to manipulate the coefficient matrix of a model and to customize the solver by using optimization callbacks also. Then, the newer object oriented modeling tools proposes to user more flexibilities in modeling and solving. OMAC is an example of these systems. Its built-in functions and callbacks allow the user to customize the internal structure of the model and the solver. In addition, its object oriented structure eases to call algorithms and solvers several times. OPL Script also has a tight integration of model and solver. It makes

it possible to state many applications that require solving several instances of the same problem, sequences of models, or a combination of both. Then, some modeling tools integrated into programming languages can be placed into the higher levels. The routines within FLOPC++ allows to make modifications on the objective function, the bounds on variables, right hand side and sense of constraints, and variable types, and to add or delete columns and rows without regeneration of the problem. In OptimJ, the manipulations over the model are immediately represented on the solver due to its extract methods generated by the OptimJ compiler, and then there is no need to reset the solver. Similarly, when any modification on an existing model is needed in Concert Technology, it does not require starting a new model and the solver from scratch due to the modification methods of the individual modeling objects.

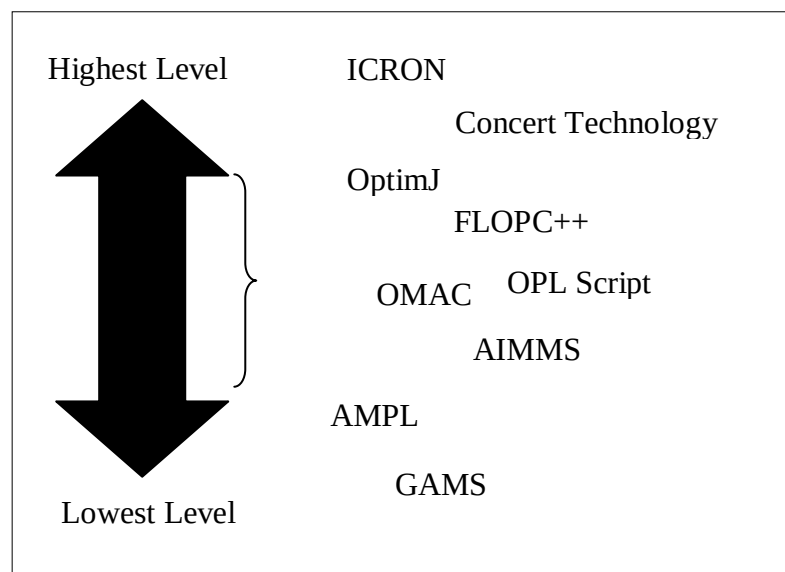


Figure 1.1. Model-solver communication tightness.

Our research is motivated by the studies on the modeling efforts in using decomposition techniques in large-scale optimization problems and the integration level between modeling tools and solvers by considering decreasing the solution time. The purpose of this work is not to develop a new solver or to change any existing solver. We proposes a new object oriented mathematical modeling tool that allows users to model large-scale optimization problems without dealing with program specific codes, and to have more feasible environment by providing closer integration between modeler and solver. Due to this tight modeler-solver relationship, the user can implement specific

techniques such as column addition, row addition, objective function change, and/or decision variable change by communicating with the selected solver in any time until the whole model is solved. It is crucial that since the modeler can reach the last output of the solver in any time and can modify it in anywhere of the algorithm, it is not needed to resolve a model from scratch after a change appears on. Thus, neither there is a need to redefine some functions to prepare the new model, nor the repetitive iterations of the solver have to be done again and again. The other significant contribution of this study is to maintain the back-translation of the solver output just after communicating with the solver in each loop. That is, the solver output can create its representation within the object world automatically whenever there is interference to the solver. Thus, the user only deals with the object-world by modeling the algorithms in terms of objects and then observing the output of the solver on the associated objects. Due to the tight integration of the modeler and the solver after this work, the user has more flexibility to model and to solve complex and large-scale problems. All of these developments have been on the optimization system called ICRON which has object orientation architecture. The previous mathematical programming structure of the system was detailed in another study (Sungur, 2003).

The rest of this work is organized as follows: there is a literature survey about large scale optimization and decomposition techniques as well as traditional and state-of-art optimization systems in Section 2. Then, the system architecture of ICRON and the new functionalities of the system are proposed in Section 3. Next, in Section 4, there are two case studies and an extension of the second case which are modeled in ICRON. Finally, in Section 5, the work is concluded.

2. LITERATURE REVIEW

The importance of large-scale optimization to solve the real life problems have led to some developments in modeling and solving techniques. Both the decomposition techniques and the tools have played crucial role to get more efficient way in optimization of these hard problems. Then, in order to better represent the direction of the improvements in the optimization tools, there have been some classifications such as matrix generators, block-schematic systems, algebraic systems, structured modeling, database systems, iconic systems (Greenberg and Murphy, 1992). As related with this work, the developments are classified as frameworks and scripts, algebraic systems, object oriented systems, and the modeling tools which are integrated into one or more programming languages.

2.1. Large Scale Optimization and Decomposition

Approximately three decades ago, mixed integer programming models were recognized as an important paradigm for solving real business problems. Since generally these problems are large-scale, and users can come upon several hundred thousands of equations and unknowns, there have been many research studies to exploit the special structures of certain models so that large models with complicating constraints or complicating variables can be solved with more simplified solution procedure. To solve these problems the use of some special techniques is either convenient or required. Decomposition techniques allow certain type of problems to be solved in a decentralized or distributed fashion so that large problems are converted into one or more appropriately coordinated smaller problems of manageable size.

The decomposition principle is a systematic procedure for solving large-scale linear programs or linear programs that contain specially structured constraints or variables (Bazaraa et al., 1990). Linear programming problems with complicating constraints involve constraints from different blocks considerably complicate the solution of the problem and prevent its solution by blocks, and the complicating variables are those variables preventing a solution of the problem by blocks, i.e., a straightforward solution of the problem. The constraints (variables in the problems with complicating variables) are

partitioned into two sets of complicating constraints (variables) and constraints (variables) having special structure. The decomposition procedure has a strategy of operating on these two separate linear programs. There is a mutual exchange of information between these programs until an optimal solution to the whole of the original problem is achieved. The three of the techniques to apply the decomposition technique in linear programming problems having special structure are Dantzig-Wolfe, Benders' decomposition and Lagrangean relaxation techniques. These techniques are mentioned shortly below.

Dantzig-Wolfe decomposition is a way of breaking a problem with special structure into an easy part, which can be solved independently, and a hard part (Dantzig and Wolfe, 1960). This technique uses column generation to create proposed solutions for the set of easy constraints. Dantzig-Wolfe decomposition technique is discussed in many texts on linear programming such as (Chvátal, 1983), (Bertsimas and Tsitsiklis, 1997), and (Dantzig and Thapa, 1997). In addition, J. R. Tebbboth has a research study on computational evaluation of Dantzig-Wolfe in general and Dantzig-Wolfe and parallel computation (Tebboth, 2001).

Dantzig-Wolfe decomposition can be contrasted to Benders' decomposition (Benders, 1962), which is based on row generation. Benders' algorithm allows us to solve a linear programming problem with complicating variables and special structure in a distributed manner. This structure often occurs in applications such as stochastic programming. The algorithm adds new constraints as it progress towards the solution and is therefore referred to as a row generation approach.

Lagrangean relaxation is one of the best relaxation techniques for the difficult optimization problems with complicated constraints. According to this technique, a subset of constraints is weighted by Lagrangean multipliers, and then added to the objective function. After that, a series of subproblems is created again, and solved with converging upper and lower bounds of the objective function (Dominguez- Ballesteros et al., 2002).

2.2. Frameworks and Scripting Languages

While algebraic systems and object-oriented systems are often used in recent applications and thus they are detailed later in this section, some of the modeling systems including frameworks and scripts are also among the new technologies that are developed to increase flexibility and efficiency of optimization systems. Several frameworks such as MINTO (Nemhauser et al., 1994), MIPO (Balas et al., 1996), BC-OPT (Cordier et al., 1999), SIP (Martin, 1998), ABACUS (Jünger and Thienel, 1998), and SYMPHONY (Ralphs and Güzelsoy, 2005) are the developments that allow users to customize solver operations. The majority of these frameworks are focused on providing an infrastructure for implementing branch-and-bound algorithms in which the user could provide their own specific methods for customizing both the branching and the bounding operations.

Of the list above, the only frameworks that provide some facility for branch-and-price, which is an algorithm based on simultaneous column and row generation, are MINTO, ABACUS, and SYMPHONY. The end-goal in designing such frameworks was to automate the most common elements of the branch-and-cut (or price) algorithm, allowing the user to focus on the problem-specific hooks (Galati, 2010). In addition, some of the frameworks (SYMPHONY, for example) were designed in a generic manner to allow greater flexibility for the user to override just about every algorithmic function.

Besides these frameworks, Decomposition for Integer Programming (DIP) (O’Sullivan et al., 2011) has been developed to tie together various algorithmic approaches related to decomposition methods in order to reduce the burden of users emanated from great flexibility. It is an open-source extensible software framework for implementing decomposition-based bounding algorithms for use in solving large-scale discrete optimization problems. The framework provides a simple API for experimenting with various decomposition-based algorithms, such as Dantzig-Wolfe decomposition, Lagrangian relaxation, and various cutting plane methods. The user can specify customizations, such as methods for generating valid inequalities and branching, in terms of the variables of the compact formulation, without having to worry about the details of any required reformulations.

On the other hand, some scripts such as OPL SCRIPT have been developed. OPL SCRIPT is a script language for composing and controlling OPL models. It makes it possible to state many applications that require solving several instances of the same problem (e.g., sensibility analysis), sequences of models, or a combination of both as in column-generation applications. OPL SCRIPT supports a variety of abstractions such as the concepts of models and abstract models that make it possible to develop, maintain, test, and reuse models independently of the scripts using them. Hentenryck and Miche, 2000, state that the ability of OPL SCRIPT to import data between scripts and models and linear programming allows it to achieve more performance in column generation applications.

2.3. Algebraic Systems

Algebraic modeling languages let the user set the variables, parameters and equations over indices to construct an algebraic form. This form is converted to a mathematical program by the algebraic system itself after it is expanded. Since the user has to use the algebraic terms to define a mathematical model, the representation of the real life objects to the mathematical world and then the back-translation of the solver output to the object world require much effort of the user.

2.3.1. GAMS

The General Algebraic Modeling System (GAMS) was the first algebraic modeling language and allows users to model complex and large-scale problems (Brooke et al., 1998). However, the models should be described in concise algebraic statements. It is connected with a large numbers of solvers such as BARON, COIN solvers, CONOPT, CPLEX, DICOPT, GUROBI, MOSEK, SNOPT, and XPRESS (GAMS Solvers, 2011).

Implementation of decomposition techniques in GAMS requires conceptually high level definition of indices, variables and dynamic sets defined in a column generation algorithm to add a new column to the model. As the complexity of the problem increases, the meaning of the sets becomes more abstract within a great number of code lines. This abstraction results in a weak representation of the real problem in the mathematical world and then hard translation of the solution. In the column generation algorithm modeled in

GAMS, for example, while the dynamic set referring to the current decision variables grows by the generation of each new column; the solver of the model is expected to start solving the model all over again by using this dynamic set and by forgetting its previous progress. The other drawback is the obligation to define an upper bound for sets. Even though the number of possible patterns and the number of iterations up to the optimum solution is found are not known actually during modeling, GAMS requires specifying an upper limit at the definition of these sets. When the optimum solution is obtained without needing to use all values of these dynamic sets, there are unused values between the value used lastly within the loop and the upperbound of the set. On the other hand, there may be insufficient value definition when the loop within the algorithm continues to find an optimum solution and needs more values although all values defined within the sets are already used before.

2.3.2. AMPL

A Mathematical Programming Language (AMPL) (Fourer, 1983) is another algebraic modeling system that similarly needs to define sets and parameters. It is possible to implement iterative schemes such as column generation, Dantzig-Wolfe decomposition, Benders' decomposition, and lagrangian relaxation by using the AMPL command language's looping and testing features. AMPL allows a model to be specified independently of the data used for a specific instance of the model so that the model, data, and run files are separated in the system. AMPL does not operate over the database but it allows creating an input data file consisting of the static data needed to solve the model. The model file, also, includes the algorithm with sets, parameters, indices, and equations. The run file, on the other hand, gathers the information of the model and the data in an intermediate form which can be read by the solver. At the end, it consists of the commands that will manage the relationship between the data, model and the solver such as the selection of solver, determining the model and data files, defining relax and presolve options, creating and controlling loops between models, and using the solution output.

2.3.3. LINDO/LINGO

LINDO was developed as a scalar-based modeling language not requiring defining sets and indices. While LINDO was very efficient to solve the small sized problems rapidly, the increasing importance of information systems to solve the large-scale real-world optimization problems led to develop newer systems (LINDO Systems, 1999). Later, LINGO algebraic modeling system was evolved to exploit these new capabilities. This new system has allowed the user to define sets as an alternative to use scalars especially in the small models.

Throughout the past years, there have been the newer versions of LINGO and each has increased the capabilities of the system. By the version LINGO 10.0 (LINDO Systems, 2009), it has allowed users to easily solve one or more models in a loop where model solutions are used as input for the subsequent models. LINGO also includes examples of using this capability to solve a cutting stock model using column generation.

2.3.4. MPL

Mathematical Programming Language (MPL) (Maximal Software, 1993) is another algebraic modeling language that allows the user to build optimization models using algebraic equations. It has an opportunity to import data directly from databases or spreadsheets, and then export the solution back into the database.

For large optimization problems, MPL has advanced capabilities in the fields of sparse indexing and data management so as to handle large amounts of data. It maintains flexibility to work with subsets of indexes, functions of indexes, and compound or multi-dimensional index sets. It also enables user to gather both indexes and data values from various data sources and import them directly into the model.

2.3.5. AIMMS

AIMMS (Roelofs and Bisschop, 1993) is a high level algebraic modeling language that enables users to implement advanced modeling concepts and build their own heuristics

with its extended algorithmic capabilities. It gives users the opportunity to work with multiple, differing instances of a single mathematical program, and to manipulate the individual rows and columns and the coefficient matrix of a mathematical program instance, for example to efficiently implement a column generation scheme. It is also possible to monitor and guide the behavior of the solver by using optimization callbacks.

The customizable open algorithm implementation in AIMMS allows modelers to fine-tune it to their specific models and use the features of Generated Math Program library likewise in the Benders' decomposition open algorithm included in AIMMS. These features drastically reduce the modeling time of the large-scale problems. Being one of the examples of large optimization problems with increasing number of possible patterns, the cutting stock problem is solved by generating the possible cutting patterns recursively in an external procedure using the AIMMS API (Bisschop, 2011). It also uses the LILE heuristic when the model is solved over and over again with the same finals, but with different demand scenarios in order to speed up the solution process. Moreover, generated math program library used in AIMMS allows for direct updates on the generated problem during loops while conventional math programs used for pattern generation approach require regeneration every time.

2.4. Object Oriented Systems

The object-oriented paradigm has been successfully used in many programming, specification, analysis, and knowledge representation tasks. This formalism is recognized for the flexibility and the support provided for designed subsystems which define abstract data types and build reusable software components. Object orientation has been also used in the formulation of the mathematical models. Among these object oriented systems, there are MME (Lahdelma, 1994), BLOOMS (Gagliardi and Spera, 1997), ODBMS (Huh and Chung, 1995), OMAC (McKay et al., 1997), TechTool (Linninger et al., 2000), AREMOS (Birgören, 1995), and ICRON (ICRON Technologies, 2010), (Sungur, 2003).

MME, which means Mathematical Modeling Environment, is an experimental, object-oriented, rule-based environment supporting mathematical modeling and algorithm development. MME provides a formal modeling and algorithm specification language,

automatic support for manipulating expressions symbolically and numerically, and built-in optimization software. Since it does not have a GUI, users have to be highly skilled in coding with its mathematical programming language which provides the procedural features of C++. Moreover, the abstract environment and formalism of modeling in MME makes the representation and the back translation of the problem more difficult.

BLOOMS is another object oriented system implemented in C++ and developed to benefit from the conceptual flexibilities of Structured Modeling. It uses the genus graph representation of Structured Modeling with nodes and arcs. An internal compiler transforms this representation to an executable C++ code and then a mathematical program is created after this code is run. After that, an internal solver is used to get the solution of this mathematical program. There is no model-data independence in BLOOMS.

ODBMS provides an object oriented database management system incorporating AMPL, GAMS, and SML. It proposes a generic modeling paradigm applicable to each of these algebraic modeling languages. By that way, its object oriented architecture provides to decompose a model into interlinked modules formulated in any of these algebraic modeling languages. Due to the property that the solver is called by the same statement for all combinations of AMLs, it satisfies the model-solver independence.

OMAC also provides a system with an object oriented architecture and a generic modeling environment in which several applications and tools are integrated so as to ease modeling of problems and to reuse the models developed before. While models are built and manipulated by using its GUI, the internal functionality of the system including the actual solution procedure is separated from this external environment. By using its built-in functions and callbacks, it is also possible to customize the internal structure of the system.

TechTool, with its object oriented architecture, allows constructing models specific to only chemical processes by using its generic mathematical language. During building models, modeling devices and links are created and their behaviors and properties are defined. This process does not require formulating any equations. Later, models are manipulated and solution procedure is implemented apart from the model building phase. It does not support the model-data and model-solver independence.

AREMOS is a system that maintains an internal object oriented model of the visual refinery model (Draman et al., 2002). Mathematical constraints and objective function of a linear program are internally generated through the objects of this model. It is an iconic system where nodes and links are used in modeling. AREMOS is an application of clone-based modeling in the refinery environment. There are studies in the literature related with clone-based modeling which allows generation of mathematical models automatically via domain specific nodes (Gerişkovan, 1999), (Köseoğlu, 1999), (Ünal et al., 2002), (Draman et al., 1995), (Draman et al., 2000).

ICRON is an object oriented optimization system which is developed in C++. Its Graphical Scheduling Algorithm Modeling System (GSAMS) allows users to construct and customize algorithms by using node and link structure instead of coding algorithms. The created algorithms seem like flow-chart with nodes representing the function source and links carrying the output data of the previous node to the next node. With its several explorers, it helps to use modeling and scheduling tools grouped for special purposes. In ICRON, every algorithm is modeled as based on object oriented data models with class definition and attributes by using its GUI. The constructed algorithms are associated with the created class definitions so that they look like the methods of the classes. The constructed algorithms can be used more than once in any other algorithm easily as a component of the algorithm. Thus, this eases to develop much more complicated models by grouping some parts of the algorithm using only GSAMS environment without dealing with programming and codes. The existing mathematical programming framework of ICRON is detailed with a study (Sungur, 2003). In this study, it is proved that linear and integer mathematical programming problems modeled in ICRON can be solved by the supported solvers; LP_Solve, GNU_LPK, COIN, and CPLEX.

This thesis improves feature of ICRON Mathematical Programming tool allows to model optimization algorithms of the large-scale problems even some decomposition techniques such as column generation. Although its previous version has some shortcomings in loops with a few modifications on the same model, it is now more efficient by not requiring using the same objects created previously in the first iteration and resolving the model. The new version of the system now eliminates the repetitions in

model and solver iterations, and so the solution process of large-scale algorithms including loops takes shorter time.

2.5. Modeling Tools Integrated into Programming Languages

Some of the tools used for the optimization of mathematical modeling problems are integrated within programming languages. They allow using the libraries and functionalities of a programming language in modeling of an optimization problem.

2.5.1. FLOPC++

FLOPC++ is the abbreviation of Formulation of Linear Optimization Problems in C++ (Hultberg, 2003). It is an algebraic modeling language implemented as a C++ class library. Within a C++ program, it serves as a declarative optimization modeling system similar to GAMS, AMPL, and AIMMS. Its syntax is close to the notation used by most of the modeling systems and uses the Optimization Solver Interface (OSI) from COIN. As the strengths of the system, the routines within FLOPC++ allow changing the objective function, the bounds on variables, right hand side and sense of constraints, and variable types, and to add or delete columns and rows without regeneration of the problem. Moreover, since modeling objects are C++ types, it is possible to implement high level modeling extensions. It also facilitates to embed linear optimization models in software applications as the traditional strengths of algebraic modeling languages are preserved.

2.5.2. ILOG Concert Technology

ILOG Concert Technology is a set of libraries offering an API for C++, Java and .NET applications and included as part of IBM ILOG CPLEX Optimization Technology (ILOG, 2008). It allows C++, Java and .NET developers to build up a high-level representation of algebraic modeling of optimization problems requiring CPLEX algorithms. Since optimization models are separated from the algorithms, problems can be decomposed into subproblems. By that way, separate parts of a large problem can be solved by using different algorithms. Concert Technology has a C++, Java, and .NET support for the CPLEX Optimizers so that it is possible to develop object oriented designs

of models. Since variables, constraints, and models in mathematical programs are represented without matrices, Concert Technology allows modifying problems easily. It is also possible to use column generation techniques by stating columns as expressions corresponding to the dual problem.

C++ applications in ILOG CPLEX Concert Technology are composed of a set of interacting modeling and solving objects. While modeling the problem, each variable, each constraint, and the objective function are represented by objects of the appropriate Concert Technology class. All modeling objects are created as related with an environment.

When any modification on an existing model is needed, it does not require starting a new model from scratch due to the modification methods of the individual modeling objects. For example in a column generation example, it is required to add columns, change coefficients in an objective function, and change the type of variables. By using extractable objects to make these modifications to the model, Concert Technology notifies the algorithms about the modification. Other details of the Concert Technology in ILOG CPLEX User's Manual offer more detail about modifications and notifications.

2.5.3. Java ILP

Java ILP is a java interface to integer linear programming solvers such as Lp_Solve, CPLEX, GUROBI, MOSEK, GLPK, SAT4J, and MiniSat (Lukasiewicz, 2008). It is designed to provide a simple object-oriented interface to ease the development of models in contrast to the other java interfaces which are adapted from C/C++ codes and provide more complicated configuration with integer variables, arrays, etc.

2.5.4. OptimJ

OptimJ (Ateji Company, 2000) is an extension of the Java programming language with language support for writing optimization models. The programming tools under Eclipse 3.2 environment supports this language.

It is both a programming and modeling language which allows to use Java objects and the whole Java library and to express all concepts found in modeling languages such as AIMMS, AMPL, GAMS, etc. Since, it is designed independently of any solver or optimization engine, it is possible to experiment with many different solvers such as LP_Solve, GLPK, GUROBI, MOSEK, CPLEX etc. without ever having to rewrite the model. In this language, a model is defined as a Java class and it can have fields, methods, and constructors just like any other class in Java. During the life cycle of a model, it is firstly instantiated which means the appropriate constructor and all the initializers are run, and the solver is instantiated. Then, the model is extracted and solved. The data of the model, on the other hand, can be read from SQL databases, Excel sheets, or XML files, and then the solutions can be sent to the application in a similar way by using the same APIs. Moreover, the problem of tracking what has been initialized or not when it is required to modify fields from an outside mechanism, for instance in modeling of column generation algorithms, is reduced by defining constructors.

In OptimJ, optimization algorithms such as column generation need only be written once and can be reused by abstracting the specific models and the concept of a pattern in a particular problem. In column generation example, there are two ways to write the algorithm with OptimJ: First, the expert can directly use the API methods like addColumn from the model written in OptimJ. Second, the expert can only decompose the original problem into master and sub-problem so that the OptimJ compiler can manage directly the new columns. Due to the extract method generated by the OptimJ compiler, each time this method is called, it incrementally adds only the new decision variables that have been created in the model since its last call, without resetting the solver (Debarbieux, 2009).

2.5.5. Xpress-Mosel / Xpress-BCL

Xpress-Mosel (Dash Optimization, 2004) is a modeling and optimizing environment where models are called from programming languages, such as VB, C/C++, Java, or C#, and they are embedded within applications using the Mosel libraries. The model is read from a simple text file, which can be modified and integrated into an application without having to rebuild the application. Mosel models can be created using the IVE visual development environment which includes a model editor, compiler and execution

environment, debugger, profiler, deployment wizards. To solve linear, integer and quadratic programming problems, Mosel uses the Xpress-Optimizer library. Mosel's modeling language is similar to the algebraic formulation of the model.

Xpress-BCL (FICO XPress Optimization Suite 7.1, 2009), on the other hand, is an object oriented interface to the Xpress-Optimizer for building mathematical programming problems within an application. It allows using a library designed for building matrices within a high level programming language (C, C++, Java, C, or VB). Although to make modifications on the model is much harder than Mosel, to execute the model takes shorter time. In addition, while Xpress-BCL is a tool used for the development of models only for problems requiring Xpress-Optimizer, such as LP, MIP, QP, and MIQP; Xpress-Mosel allows developing models for any kind of problems.

2.5.6. PuLP and Dippy

PuLP (Pulp Documentation Team, 2010) is a mathematical modeling language using Python library for modeling linear and integer programs. PuLP allows to the user to solve a MILP problem by using a variety of solvers including CPLEX, GUROBI and CBC.

The newly developed interface for advanced mixed-integer programming, Dippy, combines the Python mathematical modeling language PuLP and DIP (O'Sullivan et al., 2011). It is an extension of PuLP so that it enables the user-defined routines accessed by the DIP callback functions. The difficulty in defining a MILP problem in a low-level language, and in implementing the indexing scheme in high-level language has led to develop such a system. It allows the user to construct the model in a straightforward modeling language and to use DIP callback functions to reach the PuLP model directly. The extended callbacks of DIP can be called in Dippy when implementing customized branching, and generating customized cuts, customized columns, and the initial columns for the master problem in a column generation algorithm.

3. PROPOSED MODELING ENVIRONMENT

ICRON is an object oriented optimization system developed in C++ and has a Graphical Scheduling Algorithm Modeling System (GSAMS) allowing users to construct and customize algorithms by using node and link structure instead of coding algorithms in a programming language. The detailed information about nodes is in Appendix. The created algorithms seem like flow-chart with nodes representing the function source and links carrying the output data of the previous node to the next node. In ICRON, every algorithm is modeled based on object oriented data models with class definition and attributes by using its GUI. The constructed algorithms are associated with the created class definitions so that they look like the methods of the classes. The constructed algorithms can be used more than once in any other algorithm easily as a node itself. Thus, this eases to develop much more complicated models by grouping some parts of the algorithm using GSAMS environment without dealing with programming and codes. The mathematical programming module listed in Advanced Nodes Explorer within ICRON is detailed in Sungur (2003). It proposes to use some nodes within model mathematical programs. By extending the code of ICRON and redefining these nodes, in the scope of this work, we have developed a new mathematical programming tool so that the system provides more flexibility to adjust the constructed model during run time and more efficiency to solve looping models via a tighter model-solver communication environment.

The environment to model mathematical programming problems in ICRON has basically three main entities at the highest level. The communication between these three entities can be seen in Figure 3.1. These entities can be classified as the business model, the mathematical programming (MP) system, and the solver. “Business model” is composed of algorithms which are constructed by the user with nodes and links within the GSAMS environment which is detailed under the title “System Architecture”. Next, “MP system” is the substructure of the modeling of mathematical programming problems so as to read the constructed business model appropriately and solve it by using the selected solver. Finally, “Solver” is any solver defined appropriately within the MP system so that it can interact with the system efficiently. The user has a right to contact only with the business model during the construction and modification of a model within the GSAMS

environment. Then, MP system interacts both with the business model and the solver. The system has a model and solver independent structure. Thus, any business model including real object definitions with nodes and links can be translated into generic objects related with an MP algorithm such as objective function, decision variable, constraint etc., and then any solver can read these data with its routines supported by the system and solve the problem. After that, the output of the solver is translated back to the business model via again MP system.

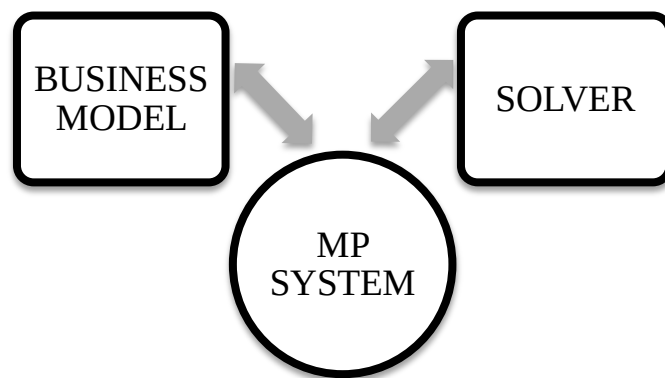


Figure 3.1. Communications between main entities of MP in ICRON.

The proposed modeling environment is based on the thought that the flow of the data from Business Model to Solver and then vice versa does not have to be once during optimization of a model. For instance, while executing models with several loops which require several calls of the solver, the flow between Business model and Solver over MP system is repeated in each loop. The proposed MP system has a flexible connection ability that it can contact with both business model and the solver during making appropriate modifications over the current model information without recreating new models.

The above abstract expression of the communication between main entities of MP during optimization of a model will be more meaningful by observing how to create an algorithm for an MP problem within GSAMS environment. Modeling of an algorithm in the system previous than this work requires creating objective function object, decision variable manager having a set of decision variable objects, and constraint manager having a set of constraints, then gathering all of them into a program object which acts somewhat as a repository for these modeling objects. Up to this point, the MP system does not interact with the solver. Afterward, at the solution phase of the model, it prepares all solver

input by using the information within this program object and then contacts with the solver to solve the given problem only once and as a whole. The communication between the system and the solver is like in Figure 3.2.a. At the end, the solution of the solver is applied to the fields of objects which are defined as decision variable. The drawback of this system appears when the model is needed to be optimized repetitively with small modifications on parameter values or constraints' orders; for example when using a decomposition technique on modeling. If so, every modeling object, even non-modified ones, and a new program object have to be created again, and then solver starts to solve from scratch, even it may have similar iterations at the beginning. The details of the system can be found in (Sungur, 2003).

As mentioned earlier, the proposed MP system can contact both with the business model created by the user and with the solver selected at the beginning of the model more flexibly at several times without recreating all modeling objects until the user terminates the model. To accomplish this functionality, it basically requires having a modeling structure with a central core object that will manage the operations between business model and the solver just like MP system's itself. The central object, called the "MP Model" object, is the basis of a model and has to be created at the beginning of each optimization process as associated with a specific solver selected by the user. Moreover, instead of decision variable managers and constraint managers in the previous system, the model object in the new system is the only repository object that holds information regarding the modeling objects as well as the solver. As long as the model object lives, it can save all the information and allows making modifications on it. Hence, unlike the previous system, MP system contacts with the solver at the beginning of the model with the creation of the model object, and the connection continues to exist over the current model object whenever a solver related job appears.

The current system allows creating more than one model object, each of which represents separate models within an algorithm. It is of high significance especially in the construction of models decomposed into several submodels. Each submodel can be solved separately with different solvers via separate model objects having different selected solver.

As mentioned earlier, MP system can contact with the solver, as can be seen in Figure 3.2.b., whenever a solver related job appears within the algorithm by using the information of the related model object. The functionality of using the model object more than one allows several update operations over its associated solver output. For example; even a model is solved before, user can add new decision variables, new objective functions, or new constraints to the model later without starting from beginning but modifying only the new model objects and the last output of the solver. The procedures of these functionalities are explained separately below.

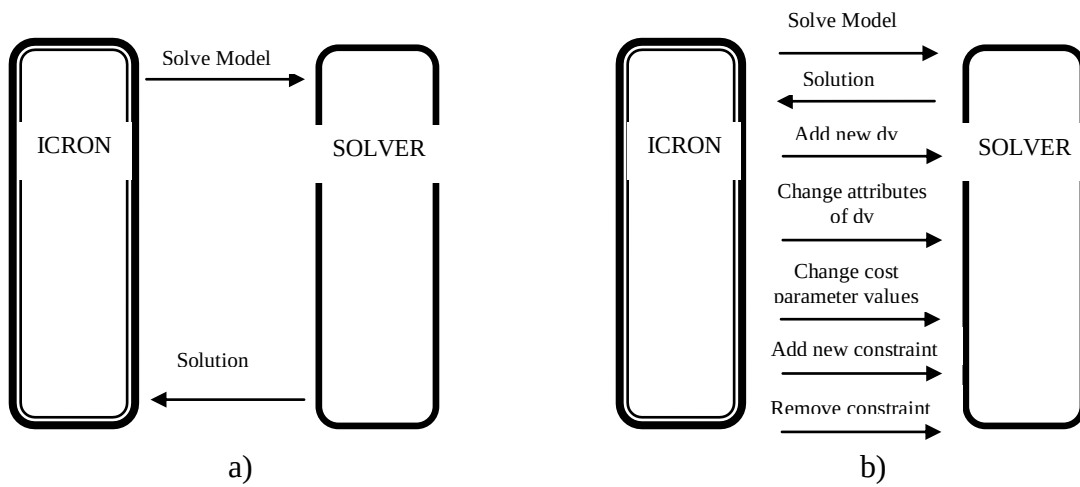


Figure 3.2. ICRON-Solver communication.

Add New Decision Variable: One of the functionalities is that the proposed system allows adding new decision variables to the last output of the solver without affecting the decision variables defined previously. If the coefficients of the new decision variable within the constraints are not known and/or will be assigned in run time, a “column” object is created by the user so as to use a temporary object to be filled with appropriate values and added to the solver output as the new decision variable and to the modeling objects as the new decision variable object. It is possible to fill the coefficient values of the column object in any way; by another optimization model, or by a heuristic, or by user-defined assignment, or by something else. This functionality is of significance especially in the construction of column generation algorithms where new decision variables are determined as based on the solution of a submodel generating column. Moreover, addition of a column as decision variable requires neither starting to prepare solver input from scratch nor creating modeling objects all over again.

Change the Attributes of a Decision Variable: The other flexibility is related with setting and updating the attributes of decision variables such as its label, type, upper bound, or lower bound values. While a mathematical programming problem is modeled, user sets attribute(s) of object(s) as decision variable. Decision variable objects have necessarily a real, integer, or binary type, and optionally lower and upper bound values. The proposed system allows user to change the type of, or bounds of the decision variables defined previously during runtime without re-creating all over again. Instead, MP system directly contacts with the solver and changes the related attribute of only the related decision variable. Moreover, it prevents to create redundant decision variables by checking the characteristics of previous ones.

Change Cost Parameter Values: Another functionality of the system is to allow changing the values of the objective function during run time without starting the solver input from scratch but only changing the objective function values. This functionality is performed exclusively by the system without requiring user to construct any checking mechanism. When the user creates an objective function, the system's itself completes necessary functions and notifies to solver as a new objective function is added.

Add New Constraint: One more significant capability of the new system is also related with the constraints of the problem. Just like defining new decision variables and objective function, the user can create new constraints even after the model is solved once. In that case, the user only creates a constraint similar to the way of creating a regular constraint. The difference that the constraint is created before or after the model is solved once is controlled by the system. It does not require the user to do any extra checking. If the constraint is added to the problem after the model is solved, the system both adds this constraint to the model by interacting with the business model and manipulates the last solver output after preparing the solver input by contacting with the solver. The advantage of this process is to gain time by not repeating the process up to the creation of new constraint. The new constraint is alone subjected to all procedures which are implemented to a constraint from creating a constraint to being a row to the solver.

Remove Constraint (Activate/Deactivate): The other flexibility related with constraints is based on the thought that all of the constraints in a problem are not expected to be used everytime. That is, the user may sometimes want to make several sensitivity

analyses by removing and then reusing some of the constraints. In that case, the first intuition would propose to construct a model without this constraint, and then to create it when needed. However, this technique is not efficient for the management of the model by the user because the business model does not imply that a constraint is excluded. If so, user would have to take notes separately for which constraint is excluded in addition to construct the business model. Besides, it would be also a burden to define the expression of the constraint in each inclusion. Therefore, the new system proposes to manage all these exclusion and inclusion operations over GSAMS environment via deactivation and activation functions. By that way, all of the constraints can be created as a part of the business model and then the ones which are expected to be out of the model are deactivated with a function without deleting the constraint objects. This deactivation process results in the relaxation of the related constraints and so these constraints do not create cuts for the feasible region of the problem.

Keeping a Reference Object for Constraints: During modeling of a problem, it is possible to create a list of constraints having the same type and same object expression by taking a list of business objects iteratively. It is critical that in large algorithms the list of these business objects may change in runtime so that they do not correspond to the right objects of the constraints. For example, there may be demand constraints iteratively created for a list of products. If the constraint does not hold the information of which product's demand it is associated with, the user cannot be sure really which constraint is created for which product's demand. Even the synchronized iterations over both constraints and products again may not be matched as the constraints are created since the products in the list or their sequences may be changed over running. In order to deal with this problem, when a constraint is created, the system proposes to set the information of the object which is related with the constraint and the user assumes it will be needed later. This object is named as "Related object". In this example, the product object can be set as the related object if the user wants to reach the information of which demand constraint is created for which product's demand.

3.1. Proposed Object Model

The proposed mathematical modeling system has simply a class relationship as can be seen in Figure 3.3. The classes in the diagram depicted in UML which are directly related with mathematical programming are MP_SYSTEM, MP_MODEL, MP_Model_Objective, MP_Model_Constraint, MP_Column, MP_Coefficient, MP_Model_Ddecision Variable, MP_Model_OptimalConstraints.

MP_SYSTEM is the main class of the mathematical programming system. It is related with MP_MODEL objects within the system and helps to initiate the execution of models during runtime.

MP_MODEL class has functions related with creation and modifications of mathematical modeling objects and the procedures for transferring the model to the format of solver input. It holds all information of modeling objects via TheObjective, pConstraints, pColumn, pDecisionVariables, and pOptimalConstraints.

MP_Model_Objective class has objects holding the expression of objective function of a model. An MP_MODEL object has any or one objective function in this system.

MP_Model_Constraint class has objects holding the expressions, types, indices, and related objects of constraints of a model. Each MP_Model_Constraint object is created for a specific MP_MODEL object.

MP_Column class has objects having attributes for objectiveCoefficient, label, lower bound, and upper bound. MP_Column objects are created for a specific MP_MODEL object. It has relations with MP_Coefficient and MP_Model_DdecisionVariable classes because creation of an MP_Column object leads to the creation of MP_Coefficient objects, and an MP_Column object may be used to create MP_Model_DdecisionVariable objects in its lifetime.

MP_Coefficient class has objects directly related with a constraint of an MP_MODEL object. The objects of this class are created when an MP_Column object is created. Objects hold an attribute of “CoefficientValue”.

MP_Model_DecisionVariable class has objects which are used as decision variables of a model and has attributes such as type, lower and upper bounds. Its objects are related with an MP_MODEL object. Its function “ApplyModelSolutionToObjects” applies the optimum values to the related fields of business objects which are defined as decision variable.

MP_Model_OptimalConstraint class has objects having attributes such as Slack, Surplus, ShadowPrice, SensitivityRangeFrom, and SensitivityRangeTo. It is used by the system to get solution information related with constraints.

3.2. Proposed Mathematical Modeling Nodes

The focus of this work is on the extension of “Mathematical Programming” nodes of ICRON and on the algorithm modeling of large-scale optimization problems. A new system in ICRON, MP_SYSTEM is defined internally to hold mathematical programming module. Mathematical programming is formulated in ICRON via Mathematical Programming Nodes. The nine MP_NODES and their running mechanisms are introduced below: “Create Model”, “Create Model Objective”, “Add Model Objective”, “Set Model Decision Variable”, “Create Model Constraint”, “Create Column”, “Add Column”, “Model Solve”, and “Apply Model Solution”. The details of running mechanisms of each node are depicted by sequence diagrams in UML.

3.2.1. Create Model

The purpose of this node, which is seen in Figure 3.4 is to create a model object of the solver which is selected using the appropriate radio button in the dialog box of the node seen in Figure 3.5. By creating a model object, it will be possible to integrate all other objects created later with the mathematical programming nodes with this model. That is,

each time an MP object such as objective, decision variable, or constraint object is created, the related MP_MODEL object will hold its information.



Figure 3.4. Create model node.

In order to create this node successfully, an appropriate input should be linked to the node, and a solver should be selected by using the radio buttons in the dialog box which appears by double clicking to the node. Although the previous version of the system supports four solvers, LP_Solve, GNU_LPK, COIN, and CPLEX, this new version currently supports only CPLEX solver to be chosen to create a model. However, it is possible to extend this structure for the other solvers by implementing the same procedure done for CPLEX.

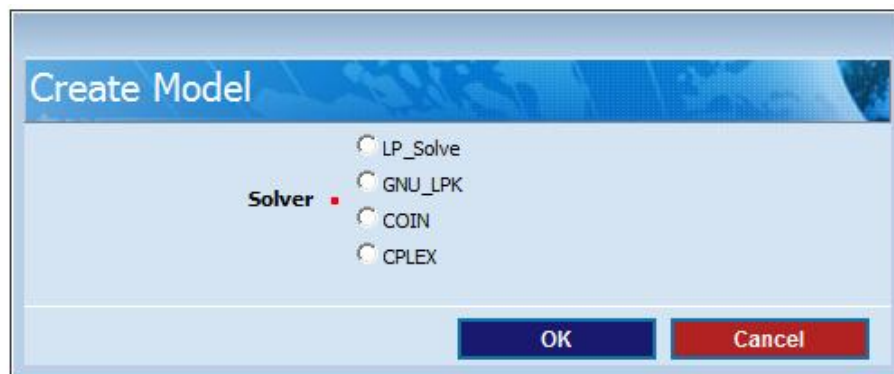


Figure 3.5. Dialog box of create model node.

An MP_MODEL object is the base for the structure of the mathematical program in ICRON. It is somewhat a repository of both modeling objects and solver objects. As soon as an MP_MODEL object exists, it allows users to make any modifications on both modeling objects and solver objects. Since MP_MODEL object is created with the solver information, it holds this information throughout its existence from beginning to the end. Due to this facility, the user can make modifications on solver information without resetting the solver but only changing the last solver output by using the solver related routines. All of these features of MP_MODEL object result in a tighter communication between model and solver.

In an algorithm modeled in ICRON, it is possible to create more than one MP_MODEL objects. This means that modeling of algorithms with several math models can be accomplished with separate MP_MODEL objects specific for each math model in the algorithm. Each MP_MODEL object has its own objective, decision variables, constraints, and solver information. This grouping of modeling objects provides the decomposition of large models into submodels, each of which can be solved with different solvers.

When the node is executed on the runtime seen in Figure 3.6, an MP_MODEL object is created and its system is set as MP_SYSTEM object. Then, this object is added to this system object and fired from the fixed output link point “MP_MODEL”.

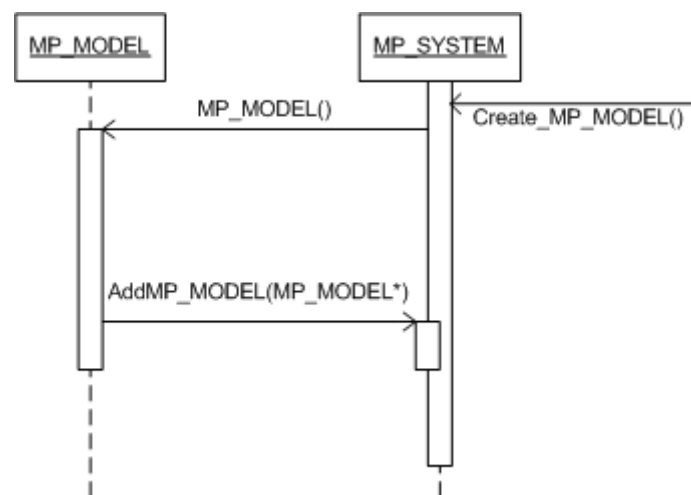


Figure 3.6. Running of create model node.

3.2.2. Create Model Objective

This node is used to create an objective function for mathematical program. Currently, ICRON supports to create a single objective function object associated with an MP_MODEL object. As can be seen in Figure 3.7, the node requires to link an MP_MODEL object as an input object so that the created objective function object will be related with the given model object. Thus, both the model and the objective function will contain the other's information.

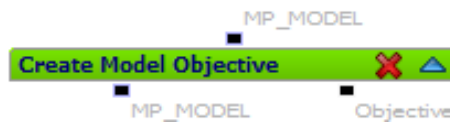


Figure 3.7. Create model objective node.

The dialog box, seen in Figure 3.8, allows the user to define the type of the objective function with a dropdown menu including type options, and to formulate the expression of the objective function via an expression builder which pops up when the user clicks on the red field next to “Objective”.

Figure 3.8. Dialog box of create model objective node.

By using this expression builder, the user can formulate an objective function in terms of dynamic input link point objects/lists of the node or global objects/lists. Then, navigation and object/list manipulation functions can be used on these input link point objects/lists. For example; as can be seen in Figure 3.9, “Create Model Objective” node has a dynamic input link point, “Products”, that gives a list of “Product” objects. Besides, product class has some attributes such as “ProductionCost” and “ProductionQuantity”.

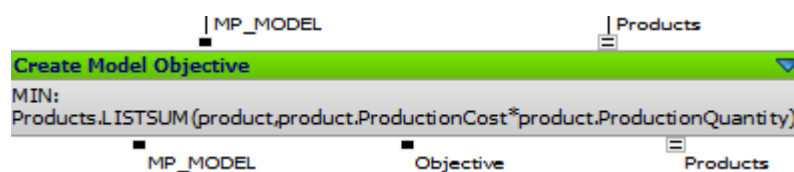


Figure 3.9. A defined create model objective node.

Then, the user wants to minimize the total production costs of all produced products. Thus, an expression for the objective function could be as in Equation 3.1.

`Products.LISTSUM(product, product.ProductionCost*product.ProductionQuantity) (3.1)`

The expression in the example denotes the multiplication of the values of “ProductionCost” and “ProductionQuantity” attributes of each product object; and by considering the whole expression, it is summation of the multiplications applied for all objects iteratively.

During building expressions of both objective function and constraints, user does not have to explicitly determine the decision variable and parameters. They are created independently from a selected decision variable. The connection between expressions and decision variables are built up during solution phase. User may determine one or any attribute in expressions as decision variable by setting the rest of them or all of them as parameters respectively. For example, in this example, while for some of the products in the “Products” list, “ProductionCost” may be a decision variable, for some others “ProductionQuantity” could be set as a decision variable. Or both of them can be parameters. However, the system does not support to set both variables as decision variable so as to prohibit a nonlinear formulation. It is also significant that the list given via the fixed input link point can be any set of objects having the same type. Hence, for example in the same example, a new products list having different size and different set of product objects can be set as input to the node in runtime without affecting the way the expression is used. This flexibility implies the model-data independence supported by the system.

When the node is executed on the runtime seen in Figure 3.10, the expression of objective function is constructed by using dynamic objects/lists in the inputs. Afterward, a new `MP_Model_Objective` object is created with the information of the given `MP_MODEL` object, an integer referring to the type of the function (“0” for minimization and “1” for maximization), and an expression which is formulated in the expression builder dialog box. This object is then set as the objective function object of the model, and finally `MP_MODEL` object is alerted to the addition of an objective function so as to use this information during the solution phase later. Then, the created `MP_Model_Objective` object is fired from the fixed output link point “Objective” and the modified `MP_MODEL` object from “MP_MODEL”.

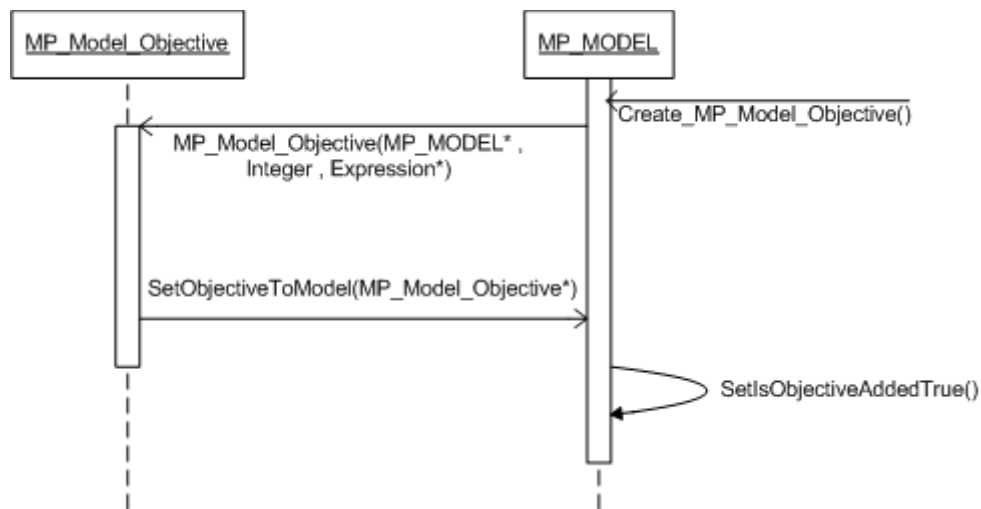


Figure 3.10. Running of create model objective node.

3.2.3. Add Model Objective

This node seen in Figure 3.11 is used to manipulate the objective function which is given as an input object called as “ObjectiveIn”. By using this node, it is possible to add additional terms to the objective function which is created by a previous call to “Create Model Objective” node. This node is not used to rewrite the given original objective function or to create a new objective function in addition to the given objective function. However, it is an option to the user that the expression of the whole objective function can be also defined in the “Add Model Objective” node. That is, the user can determine only the type of the function by using “Create Model Objective” node without writing the expression, and then use “Add Model Objective” node to express the objective function.



Figure 3.11. Add model objective node.

The additional part of the objective function is formulated in the expression builder dialog box which pops up when the user clicks “Objective” in the dialog box of add model objective node seen in Figure 3.12. User formulates the additional part likewise in the create model objective node.



Figure 3.12. Dialog box of add model objective node.

When the node is executed on the runtime seen in Figure 3.13, the expression of additional part of the objective function is constructed and added to the expression of MP_Model_Objective object in the fixed input link point “ObjectiveIn”. The expression of additional part of the objective function is constructed similarly with that of objective function and added with a concatenation operation. Then, MP_MODEL object is alerted to the change in the objective function so as to use this information during the solution phase later. Specifically, “Add Model Objective” node can be used to create objective function if only the objective function object given in “ObjectiveIn” has only the information of type of the function and does not have any expression in it. At the end, the modified MP_MODEL and MP_Model_Objective objects are fired from the fixed output link points “MP_MODEL” and “MP_Model_Objective”.

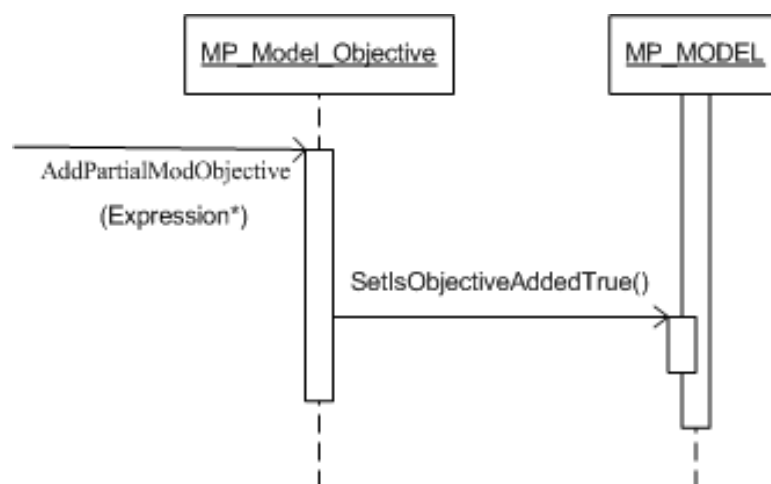


Figure 3.13. Running of add model objective node.

3.2.4. Set Model Decision Variable

The purpose of this node seen in Figure 3.14 is to create decision variables associated with the model object given with the MP_MODEL fixed input link point. The other fixed input link point, “List”, provides one or more business objects, the attribute of which is selected as decision variable. While one “Set Model Decision Variable” node allows defining one attribute of the business objects as decision variable, it is possible to set other attributes of the same objects as decision variables by successive run of this node for the same objects. Since a decision variable is associated with a model object, it can be reached later by navigating on its model object.



Figure 3.14. Set model decision variable node.

The dialog box seen in Figure 3.15 provides parameter settings related with a decision variable. At first, the red area next to “Decision Variable” indicates a field which has to be filled appropriately. When the user clicks this field, an expression builder pops up to enter the decision variable expression. It allows reaching attributes of each object which are named as iterated object in the expression builder. Then, the label of the decision variable can be defined optionally. By defining label for decision variable, each decision variable has a unique identifier including the stated label and decision variable’s own identification number next to the label. It is useful when the user wants to see the algebraic formulation of the mathematical program. Afterward, in the dialog box of the set decision variable node, there is a set of radio button required to be selected to define the type of the decision variable(s), and has a default value, “Real”. The optional fields, “Lower Bound” and “Upper Bound” are used to set upper and lower bounds of the decision variable(s).

It is important that if any object is given as input object and its attribute is set as decision variable previously, its decision variable type, lower bound, and/or upper bound can be changed by recalling the “Set Decision Variable” node with the same object and

setting the same attribute as decision variable. In that case, the new values of “DV Type”, “Lower Bound”, and “Upper Bound” override the previous values.

Figure 3.15. Dialog box of set model decision variable node.

When the node is executed on the runtime seen in Figure 3.16, initially MP_MODEL checks if there already exists a decision variable for the given attribute of the object. If the decision variable is previously created, this node modifies decision variable type, lower and upper bound of decision variable using the new values given in the node. If this is a new decision variable, then an MP_Model_DecisionVariable object is created for the given attribute of the input object with the specific type and upper and lower bounds. Consequently, the created MP_Model_DecisionVariable object is inserted into the given MP_MODEL object. This process is automatically repeated for each object within the “List”. Hence, all decision variables are owned and can be manipulated later via their model objects. Finally, the modified MP_MODEL object is fired from the fixed output link point, and the original input list from the fixed output link point “List”.

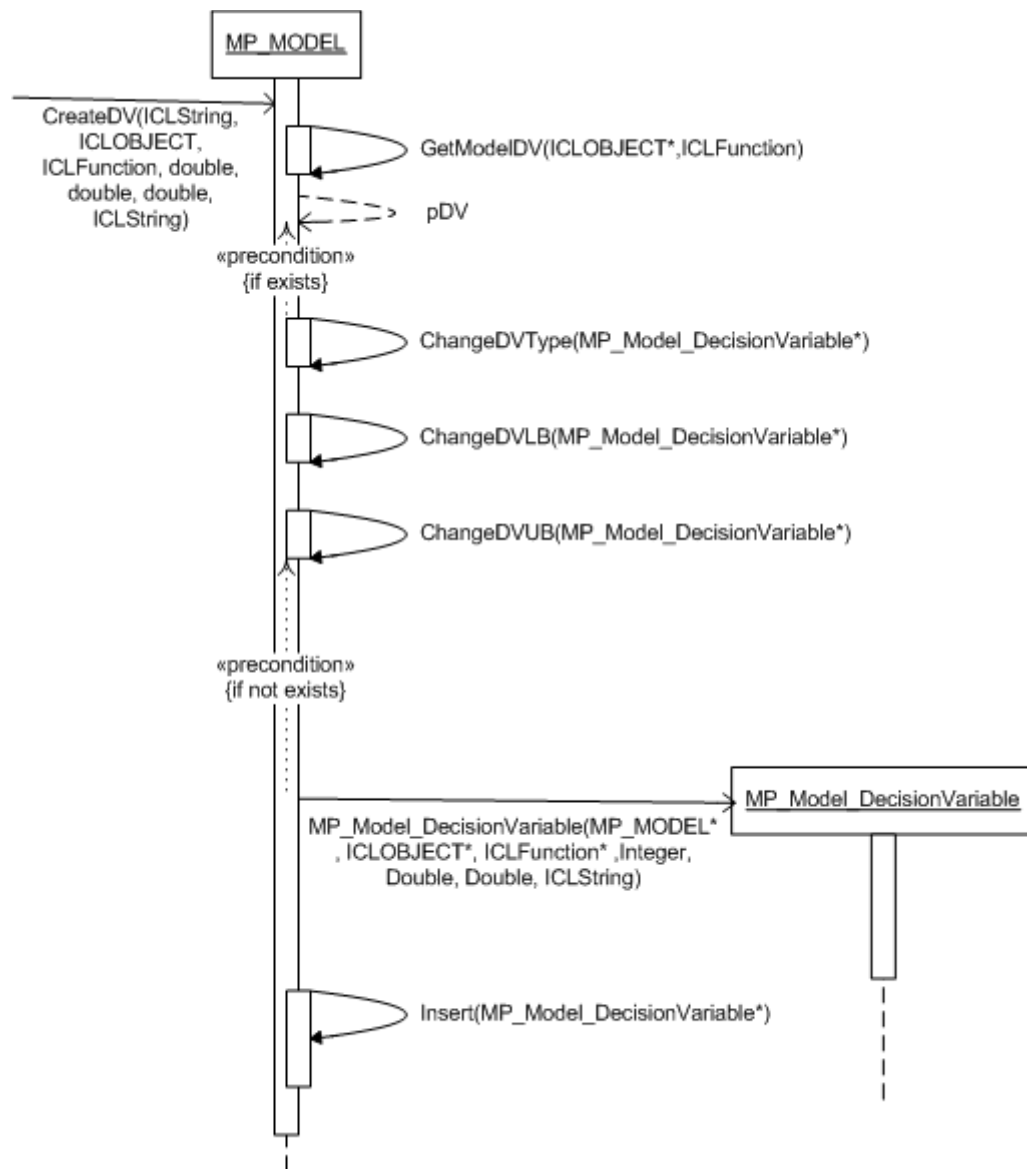


Figure 3.16. Running of set model decision variable node.

3.2.5. Create Model Constraint

This node seen in Figure 3.17 is used to create constraints associated with the model object given with the MP_MODEL fixed input link point. The system does not allow creating independent constraints. Each constraint knows its model, and vice versa. Thus, it is possible to reach constraints later via their model objects by using list manipulation routines. Although this node is designed to create one constraint at a time, it is also possible to create a group of constraints via the same node by constructing a loop structure with the appropriate nodes in GSAMS. In addition, without any significance, this node can be used in any phase of the mathematical programming; before or later the model is solved.

In each case, it is used to create a constraint. The only difference when it is called after the “Model Solve” node is that the node, additionally, modifies the last solver output by adding a related row to the matrix.



Figure 3.17. Create model constraint node.

The user links appropriate inputs to the “Create Model Constraint” node. Afterward, a dialog box of the node seen in Figure 3.18 is opened when the user double-clicks to the node. At this box, there are three fields two of which have to be filled appropriately. By using the only optional field, “Related Object”, user can specify an object related with the constraint. If it is declared, the created constraint object will know for which business object it is created. That is, it builds up a connection between the mathematical modeling constraint object and the business object. If user subsequently wants to reach later a specific attribute of an object of a specific constraint, it is possible to navigate on this constraint and its related object. The other fields, on the other hand, allow user to express constraint code and constraint expression. “ConstraintCode” field is used to give a name to the constraint by using expression dialog box. The other field, “Constraint”, is the field of where the constraint is built by using the expression dialog box. The expression process is almost similar to that of objective function. The only difference is that user must declare a right hand side, a left hand side, or a relation expression by using “ $\leq$ ” to indicate “less than”, “ $=$ ” to indicate “equal to”, and “ $\geq$ ” to indicate “greater than”. The expressions of both “RelatedObject” and “Constraint” fields are formulated with the dynamic input link point objects or lists and built-in functions of GSAMS as usual.

As an example, a demand constraint for an “ItemWidth” object can be created as follows. Firstly, an “MP_MODEL” object and the “ItemWidth” object are given as input to the node as can be seen in Figure 3.19.

Figure 3.18. Dialog box of create model constraint node.

Figure 3.19. Create model constraint node for a demand constraint.

Then, the expression of the constraint is built like in Figure 3.20. It shows that the sum of the declared multiplications should be greater than or equal to a field of an object which is “Demand” of “ItemWidth” object. Since the right-hand side value is not a constant value, and each constraint is created for each “ItemWidth” object, user can state it as a related object of the constraint. Unless related field is defined here, user cannot reach later “Demand” or other attributes, or functions of the “ItemWidth” object of a constraint if needed.

Figure 3.20. A demand constraint example.

When the node is executed on the runtime seen in Figure 3.21, left and right hand side of the expression is joined to get a single left-hand side expression. The type of the inequality is represented by an integer, 0 for “ $\leq$ ”, 1 for “ $=$ ”, and 2 for “ $\geq$ ”. Then a new `MP_Model_Constraint` object is created with the information of the model object which is given by the user with the fixed input link point `MP_MODEL`, the related object of the constraint, a string referring the code of the constraint, the expression, an integer referring to the type of the inequality, another integer referring to the index of the constraint, and a boolean referring to whether the constraint is active or deactivated/relaxed. Afterward, the created object is embedded into the `MP_MODEL` object. Therefore, each constraint is owned and manipulated later by its model object. The “Create Model Constraint” node, then, checks whether the given model has been solved before and thus whether the solver has solution information currently. Since `MP_MODEL` object holds both modeling objects and solver information, communication between these two sides are more fluent. It takes the required information from the solver, and then performs its operations; after operations, it sends appropriate data to the last output of the solver as needed. When the given model has been solved, it means that its previous constraints had been subjected to some operations during solution phase such as expansion and simplification, and then related fields of solver-matrix had been filled with the information gathered from these operations. In that case, when the new constraint is created, all these operations should be run only for this constraint and at the end, the last solver output should be modified in order to avoid repeating the whole procedure up to the new constraint is created. For that reason, the new constraint is expanded and appropriate solver input is prepared. After that, a new row representing this constraint is added below the other rows within the solver-matrix. Then, both the modified model object has one more decision variable and the modified solver output has one additional row. Finally, the modified `MP_MODEL` object is fired from the fixed output link point.

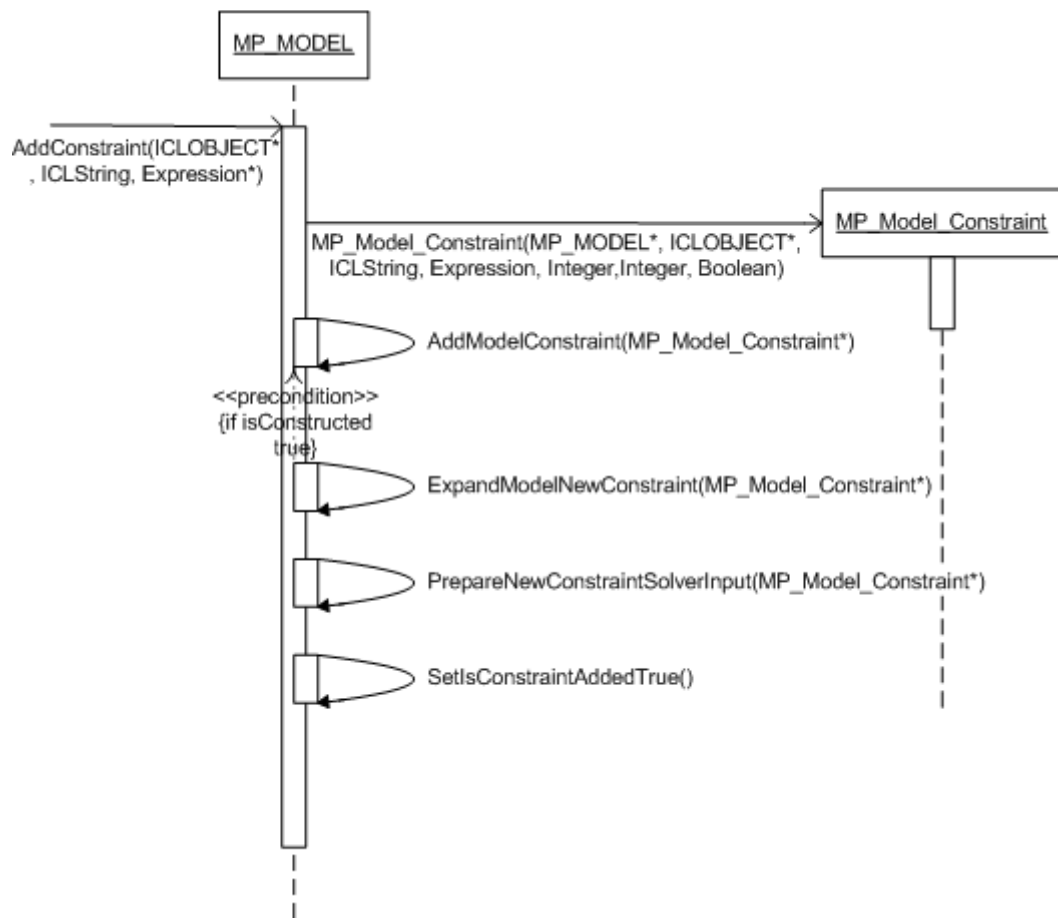


Figure 3.21. Running of create model constraint node.

3.2.6. Model Solve

The purpose of this node seen in Figure 3.22 is to solve the mathematical program provided in the fixed input link point “MP_MODEL”. During the solving stage, this input mathematical program is manipulated automatically without requiring any interaction from the user.



Figure 3.22. Model solve node.

The user links an MP_MODEL object to the “Solve” node. The node does not allow setting or editing an expression since the given model object contains all the information needed to formulate and solve the problem. When the node is executed in the runtime seen in Figure 3.23, solve function initially checks whether the given model object has been

constructed or not, which can be stated as Phase 1 and Phase 2 to make it more clear. If it has not been constructed yet (Phase 1), it means that the model has not been solved and this is the first call of the solve function for this model object. If so, first, the expressions of constraints and objective function of MP_MODEL object are expanded by using the object reference information. The expansion of constraints results in creation of new constraints which are, at this point, expanded constraints including expansion expression similar to an algebraic formulation and stored as expanded constraints separately from the original constraints. The objective function is also expanded similarly and the created expanded objective is also stored separately. The expanded expression of an objective function could be as in Equation 3.2. In fact, the expressions of expanded constraints and the expanded objective function are also simplified before the new expanded objects are created in order to settle the problem of duplications. Duplication may occur as a result of adding new terms to the objective function by using “Add Objective”. Here, some of new terms may be similar to the terms in the original objective function. On the other hand, while some of the terms may be set as decision variable and some of them may be left as parameter having constant values. Here, there may be more than one constant value that should be summed up. Thus, the simplification phase is beneficial to collect the similar variables and constant values in an expression so that there is not any duplicated terms in the final expression. The simplified expression of Equation 3.2 can be seen in Equation 3.3.

$$10*X_0 + 7*X_1 + 9*7 + 3*X_3 + \dots + 2*X_{20} + 20*3 + 4*X_0 + 6*X_1 \quad (3.2)$$

$$14*X_0 + 13 + X_1 + 3*X_3 + \dots + 2*X_{20} + 123 \quad (3.3)$$

After the expansion and simplification operations, coefficients of objective function and constraints are stored in arrays so as to be used later as an input in an appropriate format for the solver, which was selected in the dialog box of “Create Model” node. This preparation process is followed by the construction of input for the selected solver. Then, it is ready to begin the execution of “solution procedure” with the constructed input. Actually, this solution procedure refers to the process of “the given model object is constructed” (Phase 2). Therefore, while the operations in Phase 1 are only run at first call of the solve function, those in Phase 2 are run in every execution of the “Solve Model” node once the model has been constructed before. Henceforth, any modifications on the

modeling objects should be constructed individually without calling Phase 1 operations. For example, if the objective function is changed after the “Solve Model” node is called, the new objective function is prepared alone for the solver input, and then the last output of the solver is modified appropriately with the values of this objective function. Finally, the selected solver is executed and the optimal values of decision variables and of the constraints are stored in MP_MODEL object. The status of the solver execution is also stored as an integer, 0 referring to “Not Solved”, 1 referring to “Optimal”, 2 referring to “Infeasible”, 3 referring to “Unbounded”, and 4 referring to “Failure”. At the end of the execution, modified MP_MODEL object is fired from the fixed output link point.

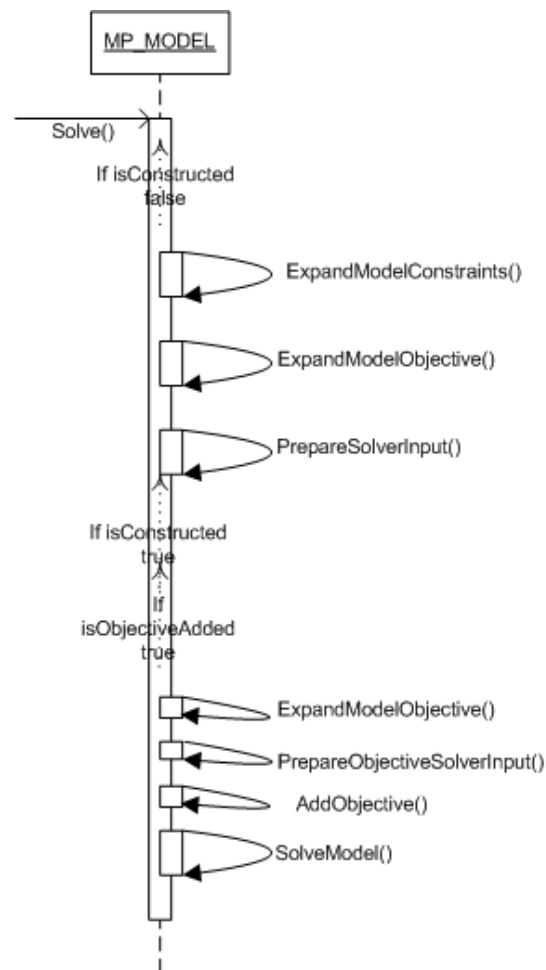


Figure 3.23. Running of solve model node.

3.2.7. Apply Model Solution

The purpose of this node seen in Figure 3.24 is to apply solutions of the mathematical program generated by the solver to the system objects. Therefore, the back translation of the mathematical programming will be supported and automated via this architecture.



Figure 3.24. Apply model solution node.

The user links the appropriate MP_MODEL object, which is solved before, to the fixed input link point of the node. After the model is solved, MP_MODEL object contains all the information about the created mathematical program, the solver, and its solutions. They are explicitly accessible via MP_MODEL object through object/list manipulation routines in GSAMS. While it is possible to assign the optimal values of some decision variables explicitly by using algorithmic formulations in GSAMS without calling “Apply Model Solution” node, the whole solution can be assigned automatically to all decision variables by using this node. The assignment of the optimal values makes it easier to translate the solver output to the real world object.

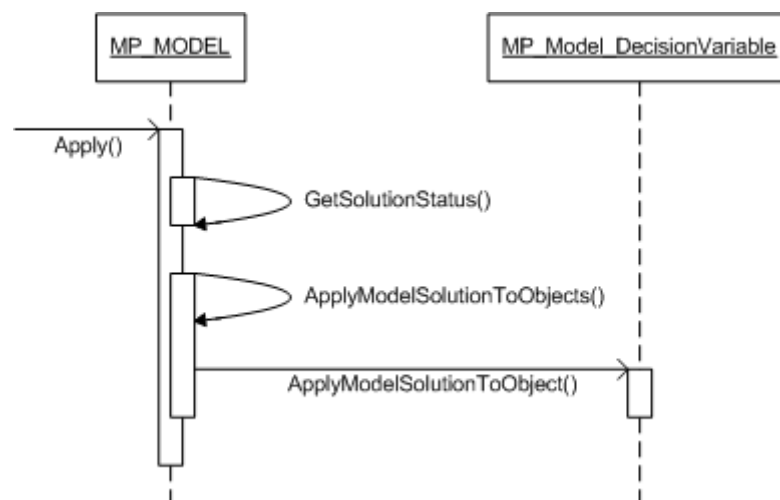


Figure 3.25. Running of apply model solution node.

When the node executed on the runtime seen in Figure 3.25, first the status of the solution is checked. If it is optimal, then MP_MODEL object iterates each decision variable, and applies the optimal solution. Since decision variable objects have a function-information to access both the object and its attribute defined as a decision variable, each decision variable automatically assigns optimal value to the related attribute of the object. Finally, the MP_MODEL object given with the fixed input link point is fired from the fixed output link point. It can be used later for other algorithmic manipulations as a runtime entity.

3.2.8. Create Column

The purpose of this node seen in Figure 3.26 is to create a column object associated with the MP_MODEL object given with the fixed input link point. Similar to the column in simplex tableau, which corresponds to a decision variable and contains the coefficients of this decision variable in objective function and constraints, the created column object is also related with a set of coefficient objects.



Figure 3.26. Create column node.

A column object is an object-oriented representation of a column in the simplex tableau. More specifically, a column object is created to hold an objective function coefficient and constraint coefficients, as many as the number of constraints in the current model object, which are to be filled later in runtime. By creating a column object, the user may use the output of the solver within the mathematical program and then, if appropriate solution is obtained, the object in the model can be converted as a solver input. That is, the created column object with empty coefficient objects and empty objective coefficient value may be filled with the optimal values of an optimization problem or with the user-defined values; then the filled column object can be added to the last solver output as a new column. Once a column object is created, its coefficient values can be changed anytime without creating a new column object if only the number of coefficient objects is

appropriate. There is only one MP_Coefficient object for each constraint and the coefficient object knows its constraint. This connection provides both great accuracy in setting the right coefficient to the right constraint, and flexibility to reach the associated object in runtime. The objective coefficient, on the other hand, is hold as an attribute of the column object not as an MP_Coefficient object.

When the node executed on the runtime seen in Figure 3.27, an MP_Column object is created as associated with the given MP_MODEL object. At this point, the column has only an index number which implies the total number of column objects that have been created since this model object is initiated. Then, MP_Coefficient objects are created one by one, by iterating on the constraints in the model, and added to the column object. Each coefficient object is associated with the iterated constraint object, and has the same index number with this constraint. At the end of the execution, the created MP_Column object is fired from the fixed output link point “MP_Column”, and the modified model object is from “MP_MODEL”.

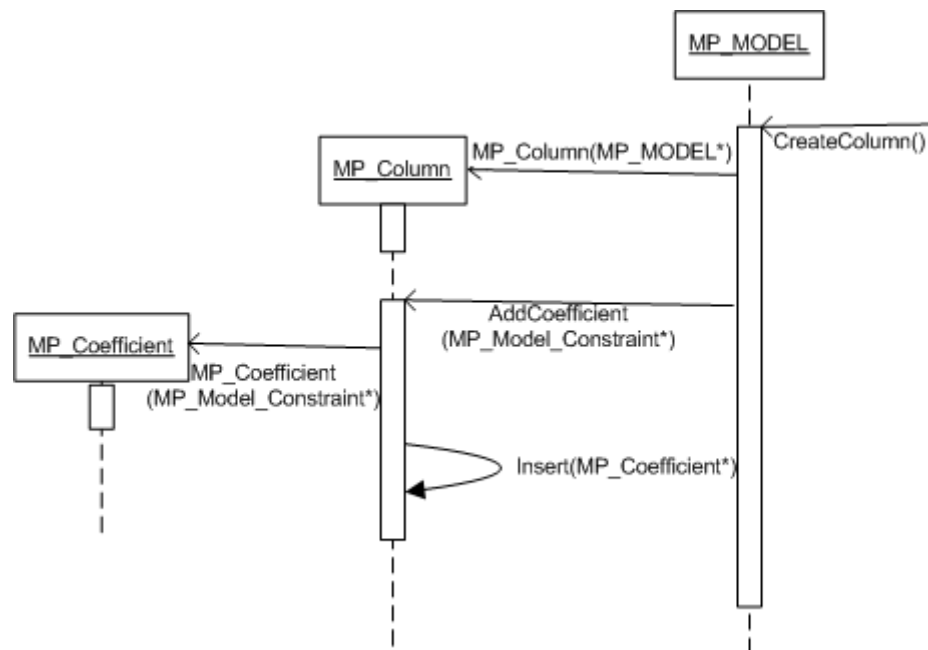


Figure 3.27. Running of create column node.

3.2.9. Add Column

The purpose of this node seen in Figure 3.28 is to make necessary modifications on both last solver output and the modeling objects as a representation of having a new full column which corresponds to a decision variable also. These modifications include addition of “New Column” to the last output of the selected solver of given MP_MODEL object, and creation of a decision variable object associated with the MP_MODEL object similarly to create a regular decision variable object in “Create Model Decision Variable” node.

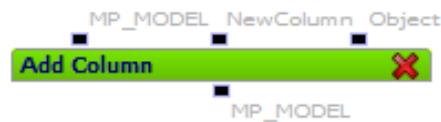


Figure 3.28. Add column node.

The user links appropriate inputs to the “Add Column” node; a model object is to the MP_MODEL fixed input link point, a column object having full coefficient values to the “NewColumn” link point, and an object indicating a real object in the model and necessarily having the same values with the column object is to the “Object” link point. To synchronize the data given to the solver and the values of its decision variable object representation, the input of “Object” is created and filled with the same values of the column in somewhere before “Add Column” node. This node does not check the synchronization between the values of “NewColumn” and “Object” objects; that is user’s responsibility. After the inputs are linked, a dialog box of the node seen in Figure 3.29 is opened when the user clicks double to the node. At this box, there are four fields which are almost the same with the dialog box of “Set Model Decision Variable” node except the radio buttons by which the type of the decision variables is selected. It does not require setting a type since it is assumed that a column added later to the model has a real type as default. Then, all other fields are filled in a similar way with the procedure in setting a decision variable. There is also a striking point in the definition of the decision variable that while objects are iteratively taken from a list and the attribute is selected on iterated object in “Set Model Decision Variable” node, in this node only a single object’s attribute is defined as decision variable so that it does not require to iterate on objects but uses the

object given with “Object” fixed input link point. The “Column Label” field is same as “Object Label” in “Set Model Decision Variable” node. By using “Lower Bound” and “Upper Bound” fields, lower and upper bounds of this single decision variable are defined respectively.

The dialog box is titled "Set column label, lb and ub". It features four input fields on the left side, each with a small colored square next to its label: "Decision Variable" (red square), "Column Label" (yellow square), "Lower Bound" (yellow square), and "Upper Bound" (yellow square). The "Decision Variable" field is a solid red rectangle, while the others are white rectangles with black borders. At the bottom right of the dialog box are two buttons: "OK" (blue) and "Cancel" (red).

Figure 3.29. Dialog box of add column node.

When the node executed on the runtime seen in Figure 3.30, firstly label, lower and upper bounds are set to the column object given with “NewColumn” fixed input link point. Then, the column with its objective coefficient value, constraint coefficient objects, and label is added to the last output of the selected solver via `MP_MODEL` object. The procedure up to this point is related with the notification of solver and updating the solver input with the new column. Although the modified model is solved accurately, the user cannot follow up the changes within the object world. Thus, there should be another procedure to represent the new column (or decision variable) as a new `MP_Model_DecisionVariable` object so that optimum value of the new column can be seen by the user like other decision variables in the model. In order to do that, user should create an object previously so as to use it just like each object within the list given to the “Set Model Objective Node”. Then, as the second procedure of the node, a new `MP_Model_DecisionVariable` object is created with the regular information. Only, the integer referring to the type of the decision variable is given as zero to make it “Real”. Finally, it is added to the model object, and the modified `MP_MODEL` object is fired from the fixed output link point “`MP_MODEL`”.

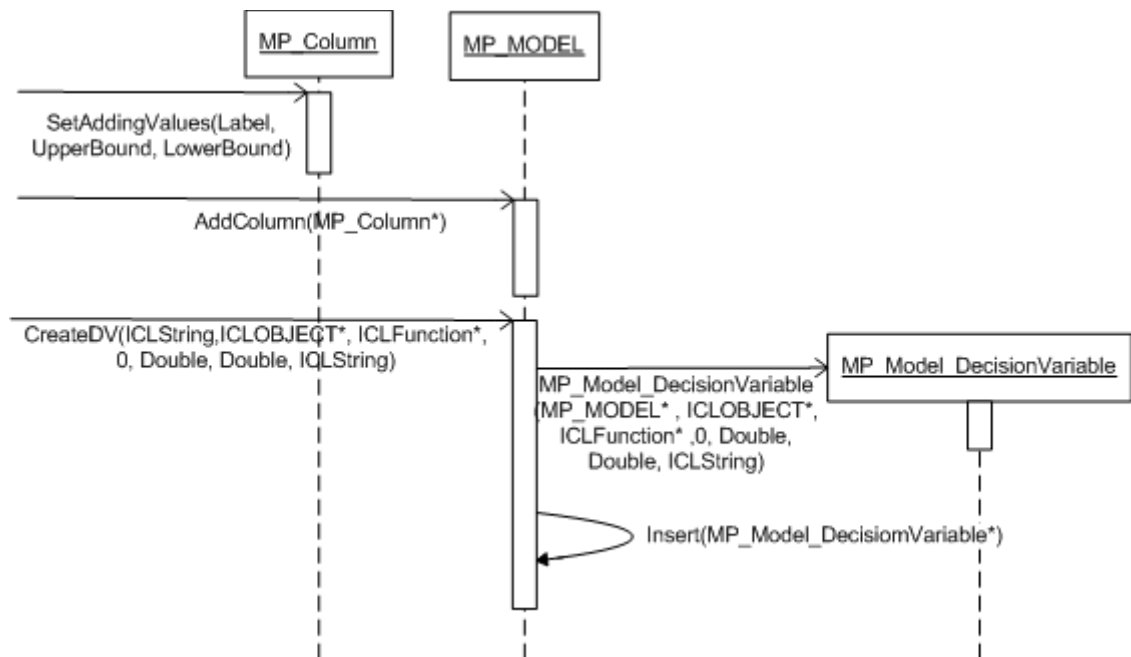


Figure 3.30. Running of add column node.

4. CASE STUDIES USING ICRON

In this study, two algorithms are modeled with the new object oriented mathematical modeling paradigm in order to describe the implementation of both small and large scale problems using ICRON, and to exemplify the functionalities of the system for the large-scale optimization problems. The well-known cutting stock problem is used as an example. The first algorithm is an introductory example of cutting stock problem, which has a single optimization model with a set of decision variables. The second one is the cutting stock problem using column generation algorithm where new decision variables can be added to the model by several iterations and thus the model may grow dynamically in size. In this example, the cutting stock problem is used to show how a problem-specific decomposition algorithm can be built in ICRON. At the end of the section, there is an extension of the second case where a new constraint is added to the problem.

4.1. An Introductory Example: Cutting Stock Problem

This introductory example is organized as the definition of the problem, giving a numerical example, and the modeling of the example in ICRON. At the end, the solution of the system is also demonstrated.

4.1.1. The Definition of a Cutting Stock Problem

The cutting stock problem is an integer linear programming problem which arises from many applications in industry. Generally, it includes a supply of rolls of material of fixed length as called the stocks and strips that are cut in different lengths from these stocks. Each combination of strips on one stock constitutes a pattern. The problem is to use as few patterns as possible to satisfy demands of strips (Kalvelagen, 2002).

A cutting stock problem can be harder to solve with a longer roll length and smaller strip length. Increasing number of combinations means increasing the quantity of patterns which are the variables in the problem. Figure 4.1 shows two different patterns from a roll of stock. Here, in pattern1 one strip with size1 and one strip with size2 are cut from one

roll of stock. Pattern2 is also created by cutting three strips with the same size 2. Except these patterns, different patterns can be constructed by combining different cutting ways. Even though there can be an exponential number of possible patterns, for any given demand, a limited number of patterns will be sufficient.

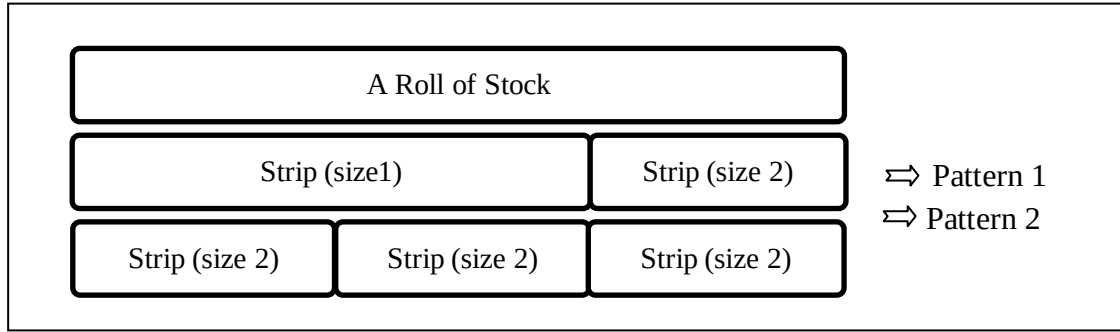


Figure 4.1. Different patterns on a roll.

A general cutting stock problem is described by using a variable to define the number of times that pattern j will be used (i.e. X_j), and another parameter (i.e. A_{ij}) indicating the number of product of length i that is the result of applying pattern j needed to satisfy D_i which is the demand for product with length i . The objective function seen in Equation 4.1 is to minimize the sum of the patterns used. Equation 4.2 is the demand constraint iterated over two sets i and j , which represent the products and the patterns. A separate demand constraint can be written for each product with length i so that the total quantity of the used product with length i will be greater than or equal to the demand of product with length i .

$$\text{Minimize } \sum_j X_j \quad (4.1)$$

$$\text{Subject to } \sum_{ij} A_{ij}X_j \geq D_i, \quad X_j \in \{0, 1, 2, \dots\} \quad (4.2)$$

4.1.2. An Example of Cutting Stock Problem

Let us consider the data from (Kalvelagen, 2002). The cutting stock problem instance can be described as follows. The rolls of stock to be cut are 100 inch wide and the demand data for four strips having different widths is shown in Table 4.1.

Table 4.1. Demand data

Width(inches)	Demand
45	97
36	610
31	395
14	211

All possible cutting patterns enumerated according to the demand data, the widths of strips, and the width of a roll. Then, 37 possible patterns are obtained as in Table 4.2 :

Table 4.2 All possible patterns

Index		ItemWidth1	ItemWidth2	ItemWidth3	ItemWidth4
0	Pattern1	2			
1	Pattern10		2		2
2	Pattern11		2		1
3	Pattern12		2		
4	Pattern13		1	2	
5	Pattern14		1	1	2
6	Pattern15		1	1	1
7	Pattern16		1	1	
8	Pattern17		1		4
9	Pattern18		1		3
10	Pattern19		1		2
11	Pattern2	1	1		1
12	Pattern20		1		1
13	Pattern21		1		
14	Pattern22			3	
15	Pattern23			2	2
16	Pattern24			2	1
17	Pattern25			2	
18	Pattern26			1	4
19	Pattern27			1	3
20	Pattern28			1	2
21	Pattern29			1	1
22	Pattern3	1	1		
23	Pattern30			1	
24	Pattern31				7
25	Pattern32				6
26	Pattern33				5
27	Pattern34				4
28	Pattern35				3
29	Pattern36				2
30	Pattern37				1
31	Pattern4	1		1	1
32	Pattern5	1		1	

Table 4.2 All possible patterns. (Contd.)

33	Pattern6	1			3
34	Pattern7	1			2
35	Pattern8	1			1
36	Pattern9	1			

At the end, there is a cutting stock problem in which the length of a roll, the widths and demands of strips (“items” as we call in this work), and the patterns are all known. The only unknown thing is the quantity of used patterns in optimum solution. The problem is finding the quantity of pattern that minimizes the total quantity of patterns by satisfying the demands of items.

Figure 4.2 represents a class diagram of the relevant business objects of the problem. There are four classes including the information of business objects: “CuttingStockSystem”, “ItemWidth”, “PatternDetails”, and “Pattern”. Since this problem includes only one roll’s width, a roll class could be associated with the problem in an abstract world so that the single object of that class represents the cutting stock problem. By considering this relation, the roll class is named as “CuttingStockSystem” with the attribute of “Length_to_Cut” representing the fixed length of the roll. Each cutting stock problem has also items with different sizes as seen in Table 4.1. To hold these items in the system, “ItemWidth” class is created with the attributes of a unique “ItemWidth_ID”, “ItemWidth_Inch”, and “Demand”. This class’s name comes from the significance of different widths to categorize items. Then, as seen in Table 4.2, there is a many-to-many relationship between patterns and widths of items. Therefore, an association class named as “PatternDetails” is created with the attribute of “Item_Quantity” so as to hold the information of how many items of a specific width will be required to be cut for a pattern. Then, containing all possible patterns, the “Pattern” class is created with the attribute of “Pattern_Quantity” which indicates the quantity of patterns.

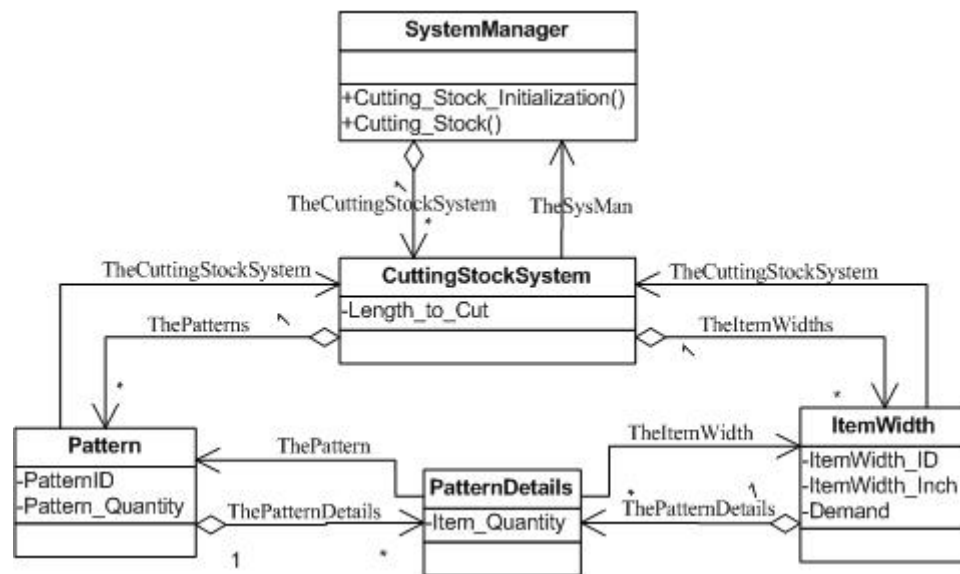


Figure 4.2. Class diagram of a cutting stock problem.

4.1.3. Modeling with ICRON

At the beginning of the modeling in ICRON, a data model appropriate to the class diagram of the problem seen in Figure 4.2 is constructed. By following the instructions about how a model is created in “Models” explorer mentioned above, a data model is created by considering the attributes of and the relationships between the classes. It can be seen in Figure 4.3. This is a completely object oriented data model providing one-to-one mapping to real world objects. There is “Cutting_Stock1” model defined in Figure 4.3. Beside the classes “CuttingStockSystem”, “ItemWidth”, “PatternDetails”, and “Patterns”, there is also an extended class “System Manager”. “TheCuttingStockSystem1” attribute in the “SystemManager” class holds the cutting stock system object so that it will be possible to reach this object and its associations over the system manager even from other models listed in “Models” explorer. Additionally, with the attribute of “TheSysMan”, “CuttingStockSystem1” class allows reaching other models from this model. The other two attributes of the class are the lists of patterns and itemwidths which are the object lists containing the whole information about its related object. Likewise, the attribute of “TheCuttingStockSystem1” of “ItemWidth” class also contains the whole information about the object of CuttingStockSystem1 class. These lists and objects represent the relations between classes such as the one-to-many relationship between

CuttingStockSystem and ItemWidth classes. Then, the relationships between other classes are also constructed in the similar way.

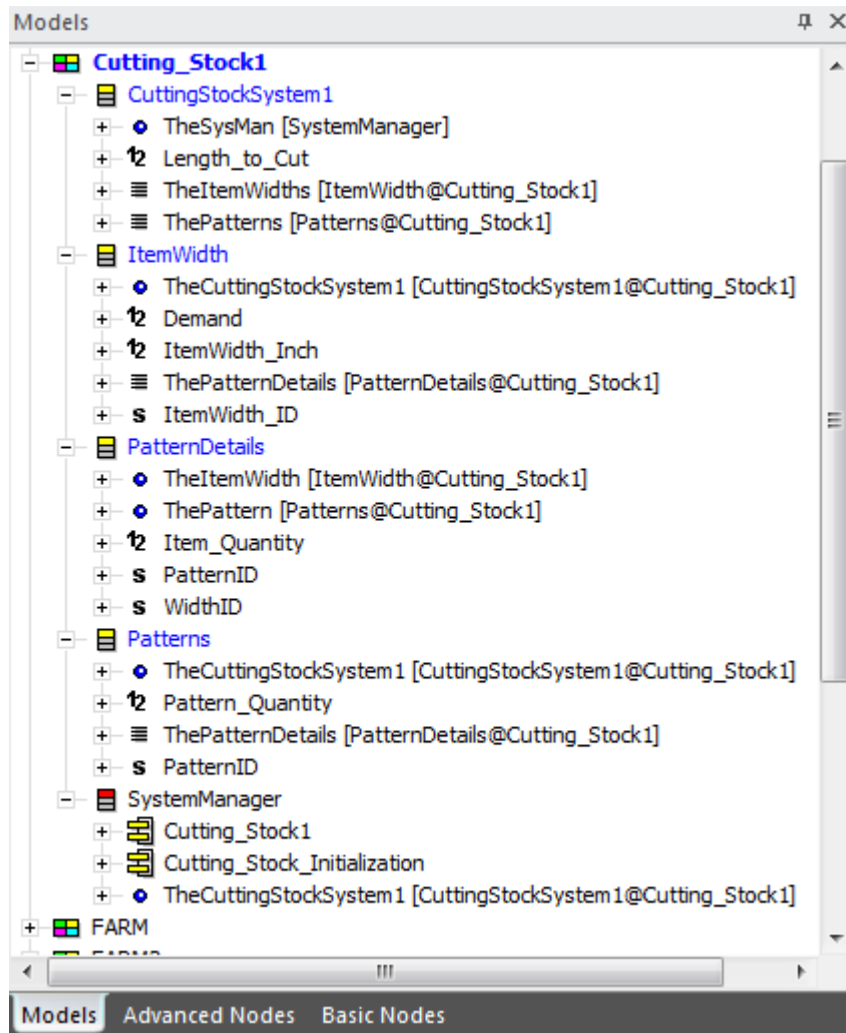


Figure 4.3. Data model of cutting stock problem in ICRON.

After constructing the data model of the problem, the problem should be initialized by providing real data so as to generate a problem instance. The data can be retrieved from a data source and/or generated in runtime. In this example, an initialization algorithm is modeled to create related problem data at the beginning of the execution. This “CuttingStock_Initialization” algorithm of the class “SystemManager” can be seen in Figure 4.4.

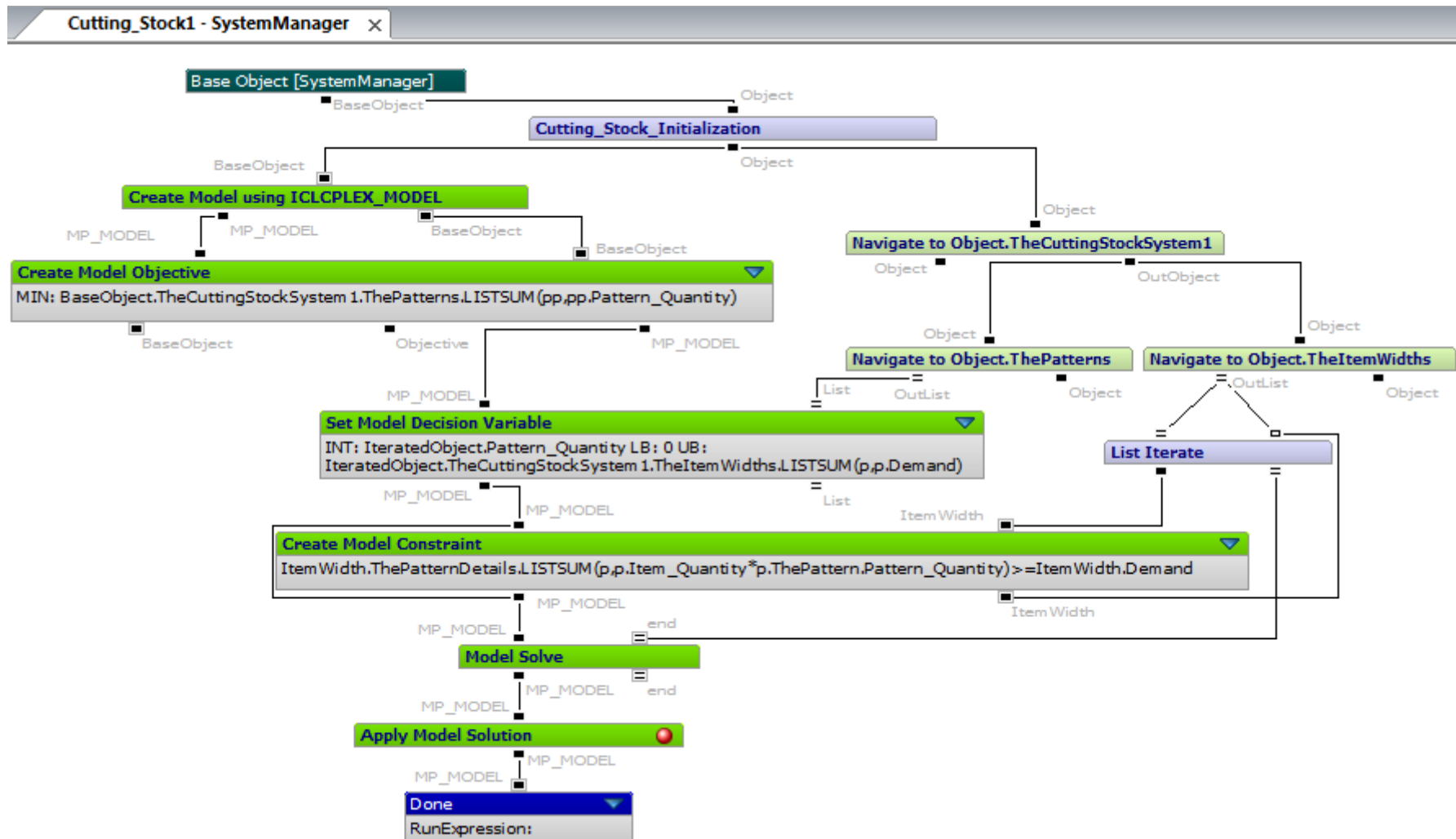


Figure 4.4. Mathematical model of cutting stock problem in ICRON.

The initialization algorithm is basically designed to generate the actual data of the cutting stock object, item objects, pattern objects, and pattern details objects. How the fields of the objects are filled with the actual data is beyond the scope of this work. Finally, this algorithm model used as a node in “CuttingStock1” algorithm fires “SystemManager” object manipulated during initialization.

The mathematical model is formulated independently from the data. Whatever data is in the objects, a mathematical programming model in ICRON starts with creating a model object specific for only this model. This is achieved by using “Create Model” node in Figure 4.4 and selecting the solver to solve the model by using the information in the model object later. Then, the objective function, the constraints and the decision variables of the model are defined using “Create Model Objective”, “Create Model Constraint”, and “Set Model Decision Variable” nodes. The expressions of objective function and constraints that can be seen respectively in Equation 4.3 and Equation 4.4 are generic and do not contain any data specific information. The formulations are based on the class definitions of the four business classes stated above. Instead of using mathematical notation such as X_s , Y_s and so on, only objects and lists are used in modeling of the problem. In addition note that, there are no indices or sets in expressions. The actual values in the objects are not embedded into the expressions. The LHS of the expression in Equation 4.4 referring to demand constraint firstly navigates to the list of pattern details objects over ItemWidth objects so as to reach only pattern details objects associated with the given ItemWidth object. For each ItemWidth object iterated over the list of ItemWidth objects with “List Iterate” node, a separate demand constraint is created over the iterated object. The navigation between the related classes with (.) operator like in these expressions can be also done with “Navigate” and “List Iterate” nodes. Then, the constraint multiplies “Item_Quantity”, attributed of each pattern detail in the list, with “Pattern_Quantity”, attributed of the pattern which each pattern detail in the list belongs to. Then, all of the multiplications are summed in the “LISTSUM” function. This will correspond to the total quantities of items used by the patterns. At that step, there is no need to know the actual values of quantity of items. Additionally, the formulation of the constraint is not affected by the number of objects in the pattern details list. This structure makes it possible to keep the formulation as is even if different pattern details are

associated with different items. The RHS of the expression is the demand of the item. It is obtained from the “Demand” attribute of the ItemWidth object.

Objective Function:

$$\text{BaseObject.TheCuttingStockSystem1.ThePatterns.LISTSUM(pp,pp.Pattern_Quantity)} \quad (4.3)$$

Constraints:

$$\text{ItemWidth.ThePatternDetails.LISTSUM(p,p.Item_Quantity*p.ThePattern.Pattern_Quantity)} \geq \text{ItemWidth.Demand} \quad (4.4)$$

Generally, decision variable selections can be done in anywhere after the model is created. The expression formulations are completely independent from the definition of the decision variables. In the above mathematical model, it is done between the objective function and the constraints. Even if the decision variables are selected before the objective function or after the constraints, there will be no change in the constructed algorithm. “Pattern_Quantity” attribute of each pattern is set as a decision variable. This decision variable is of “Integer” type with zero (0) lower bound and an upper bound of total demands of items.

The created objects under the system manager can be reached by using navigation nodes. The hierarchic structure of the class relations allows accessing both single object and a list of objects. The “CuttingStockSystem1” object is obtained by giving a system manager object to the node. The lists of patterns and of items are also obtained by giving the “CuttingStockSystem1” object.

4.1.4. Solution of Cutting Stock Problem

When the model is solved, firstly the expressions in the objective function and in the constraints are expanded and then simplified. The simplified expressions can be observed in the mathematical model which can be seen Table 4.3. In the objective function, the Pattern_Quantity attribute of all of the patterns in the list under the cutting stock system is summed up. The index number beside the attribute is added automatically by the system.

The numbers beginning with zero are assigned sequentially to each pattern object in the list. Since the patterns are sorted like the Table 4.2 during the initialization of the system, the index numbers are assigned like in the first column of this table. That is, “Pattern_Quantity_0” refers to the quantity of Pattern1, “Pattern_Quantity_1” to the quantity of Pattern10, and so on. The single demand constraint in the model is created four times for each item object. The values in the constraints are the parameters of the original formulations. They are the item quantities in this example. The right hand side value is also subtracted from the expressions of constraints, and becomes zero.

Table 4.3. Mathematical model of cutting stock problem.

Type	Z row		
MIN	Pattern_Quantity_0+Pattern_Quantity_1+.....+Pattern_Quantity_36 + 0		
Name	LHS	Type	RHS
demand constraint_0 _Expanded	2*Pattern_Quantity_0 + Pattern_Quantity_11 + Pattern_Quantity_22 + Pattern_Quantity_31 + Pattern_Quantity_32 + Pattern_Quantity_33 + Pattern_Quantity_34 + Pattern_Quantity_35+ Pattern_Quantity_36 - 97	<=	0
demand constraint_1 _Expanded	2*Pattern_Quantity_1+2*Pattern_Quantity_2+ 2*Pattern_Quantity_3+Pattern_Quantity_4+Pattern_Quantity_5 + Pattern_Quantity_6+Pattern_Quantity_7+Pattern_Quantity_8 + Pattern_Quantity_9+Pattern_Quantity_10+Pattern_Quantity_11+ Pattern_Quantity_12+Pattern_Quantity_13+Pattern_Quantity_22 - 610	<=	0
demand constraint_2 _Expanded	2*Pattern_Quantity_4+Pattern_Quantity_5+Pattern_Quantity_6+ Pattern_Quantity_7+3*Pattern_Quantity_14+ 2*Pattern_Quantity_15 + 2*Pattern_Quantity_16+ 2*Pattern_Quantity_17+Pattern_Quantity_18+ Pattern_Quantity_19 + Pattern_Quantity_20 + Pattern_Quantity_21 +Pattern_Quantity_23+ Pattern_Quantity_31+Pattern_Quantity_32 - 395	<=	0

Table 4.4. Mathematical model of cutting stock problem. (Contd.)

demand constraint_3 _Expanded	$2*Pattern_Quantity_1 + Pattern_Quantity_2 +$ $2*Pattern_Quantity_5 + Pattern_Quantity_6 +$ $4*Pattern_Quantity_8 + 3*Pattern_Quantity_9 +$ $2*Pattern_Quantity_10 + Pattern_Quantity_11 +$ $Pattern_Quantity_12 + 2*Pattern_Quantity_15 +$ $Pattern_Quantity_16 + 4*Pattern_Quantity_18 +$ $3*Pattern_Quantity_19 + 2*Pattern_Quantity_20 +$ $Pattern_Quantity_21 + 7*Pattern_Quantity_24 +$ $6*Pattern_Quantity_25 + 5*Pattern_Quantity_26 +$ $4*Pattern_Quantity_27 + 3*Pattern_Quantity_28 +$ $2*Pattern_Quantity_29 + Pattern_Quantity_30 +$ $Pattern_Quantity_31 + 3*Pattern_Quantity_33 +$ $2*Pattern_Quantity_34 + Pattern_Quantity_35 - 211$	<=	0
-------------------------------------	---	----	---

The result of this optimization problem can be seen in Table 4.5. Among 37 pattern objects, only 4 of them have positive values. The optimum policy is to use Pattern1 for 49 times, Pattern10 for 106 times, Pattern12 for 100 times, and Pattern13 for 198 times so that demands of items in each width are satisfied with minimum total quantity of used patterns. The total quantity is 453 in this example. The upperbound of the decision variable is the sum of the demands, 1313.

Table 4.5. Optimal solutions of the cutting stock problem.

Optimal Objective Value			
453			
Name	Optimum Value	Lower Bound	Upper Bound
Pattern_Quantity_0	49	0	1313
Pattern_Quantity_1	106	0	1313
Pattern_Quantity_3	100	0	1313
Pattern_Quantity_4	198	0	1313

The way of generating this mathematical program and obtaining the optimal solution is hidden from the user. By applying solution to the objects, optimal solutions can be viewed in the “Pattern_Quantity” attributes of each pattern objects.

4.2. Cutting Stock Problem Using Gilmore-Gomory Column Generation Algorithm

Solving the cutting stock problem with its general formulation mentioned above where all possible decision variables (patterns in this example) are considered during solution is not applicable in general because the number of possible patterns is in general very large. For example, if there are 10 different strip-sizes which are roughly 1/10 of the width of the roll, there will be nearly 10^{10} types of patterns, and thus that many decision variables. In our first example, even it is extremely small with four final widths; there are 37 possible patterns as can be seen in Table 4.2. Such a formulation might not even fit in memory on a reasonably large computer. Moreover, most of those patterns would not be used in an optimal solution. These considerations make column generation an interesting approach for this problem.

In this second algorithm, a column generation technique named as Gilmore-Gomory Algorithm (Gilmore and Gomory, 1963) will be applied to the cutting stock problem so as to exemplify the usage of a decomposition technique in ICRON. The aim is to speed up the solution process of the cutting stock problem by dealing with only the variables (patterns) which are more beneficial to converge to the optimal solution instead of considering the all variables (patterns).

4.2.1. Column Generation Algorithm

Column Generation is a technique of solving large-scale mathematical problems by creating loops between a small, manageable block so called master problem and a sub-problem including especially complicating constraints. During the solution of the problem, at first, master problem is solved and its solution is analyzed to define the objective function of the sub-problem. That is, dual prices for each of the constraints in the master problem are gathered from the solution of the master problem and they are used in the objective function of the sub-problem. Then, when the sub-problem is solved if it has an optimum solution with negative objective function value, it means a new column is generated and it will be added to the master problem. The new master problem is resolved with this new column. This process repeats until it finds out an acceptable solution to the

whole problem. The communication between master problem and sub-problems are as in Figure 4.5.

In column generation, the decomposition of the original problem into sub-problems and working structures of these problems depend on the fact during the pricing phase of the simplex method. Since the solver in simplex method does not need access to all the variables of the problem simultaneously, the model of the problem can also decomposed so that only a particular subset of the constrained variables, which are columns, can be solved within a sub-problem. Thereafter, if needed, other variables are accessed according to their reduced costs which are calculated by solving another sub-problem. If there is not any acceptable reduced cost, then the current solution is optimal.

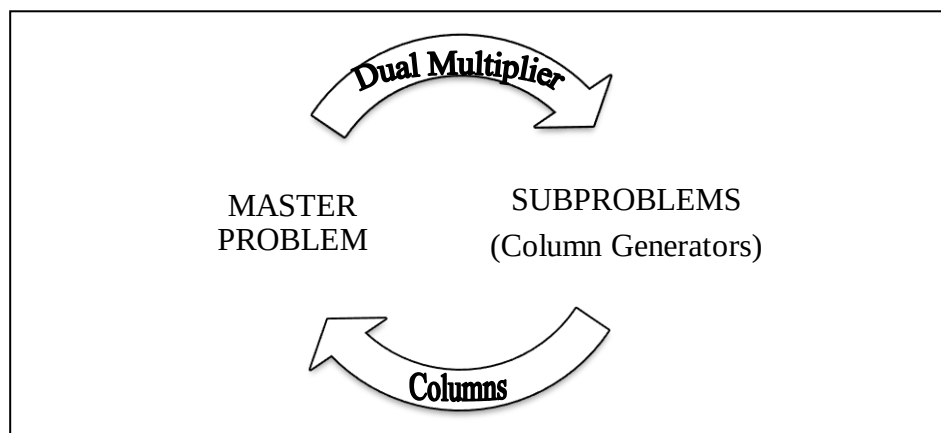


Figure 4.5. Communication between master problem and sub-problems.

A classic example of column generation algorithms is the cutting stock problem which is an NP-hard problem. Since column generation is a powerful technique for solving large-scale problems to optimality or to near optimality, many research studies have been done in order to improve its usability within different problem types. For example, firstly, Ford and Fulkerson (Ford Jr. and Fulkerson, 1958) in 1958 suggested column generation technique in the context of a multi-commodity network flow problem. Since the number of variables in this problem is too large to be dealt with explicitly, they proposed to treat non-basic variables implicitly by using several applications of a combinatorial algorithm for finding a shortest chain joining a pair of points in a network.

Gilmore and Gomory (Gilmore and Gomory, 1963) demonstrated effectiveness of column generation in speeding up the solution process of a cutting stock problem. Their approach is guaranteed to converge to the optimal solution without needing to enumerate all the possible patterns in advance. Since this method has a limitation on handling integrality by sometimes offering fractional solutions, it may result in sub-optimal or infeasible solutions. To handle this limitation, new algorithms have been developed later (Goulimis, 1990), (Carvalho, 1998). Later on, vehicle routing, crew scheduling, and other integer-constrained problems have motivated further research into column generation. More details can be found in the book specialized for column generation (Desaulniers et al., 2005).

4.2.2. Gilmore-Gomory Algorithm

The Gilmore-Gomory algorithm, which shows the effectiveness of column generation in cutting stock problem, proposes to start with a small initial set of variables instead of enumerating all possible patterns j . A small initial set is created simply by dividing the roll width with the length of each product successively. The floor value of one division gives the only nonzero value belonging to the row related with the constraint of the dividing product and to the column representing decision variable which is the pattern. The initial set, for example in the model with three constraints, will be as in Figure 4.6.

	Pattern1	Pattern2	Pattern3
Product1-width1	$\left\lfloor \frac{\text{Roll width}}{\text{width1}} \right\rfloor$		
Product2-width2		$\left\lfloor \frac{\text{Roll width}}{\text{width2}} \right\rfloor$	
Product3-width3			$\left\lfloor \frac{\text{Roll width}}{\text{width3}} \right\rfloor$

Figure 4.6. The initial set.

This method produces a feasible solution to the problem, but it may not necessarily give a satisfactory answer. To move closer to a satisfactory solution, the application can then generate other columns. That is, other decision variables (other X_j) are chosen to add to the model. Those decision variables are chosen by the sub model according to their

favorable reduced cost. The sub-problem is defined to identify the coefficients of a new column of the master problem with minimal reduced cost.

Let π_i denote the vector of the dual variables of the constraints in the master problem, the sub-problem is defined like this:

$$\text{Minimize } c_i - \sum_i \pi_i A_i \Rightarrow 1 - \sum_i \pi_i A_i \quad (4.5)$$

$$\text{Subject to } \sum_i W_i A_i \leq W, A_i \in \{0, 1, 2, \dots\} \quad (4.6)$$

where W is the length of a roll of stock and the entries A_i are the decision variables of the sub-problem and implies the number of times item i is selected in the new pattern. The objective function seen in Equation 4.5 is to get the minimum reduced cost. The general formulation of a reduced cost is the subtraction of total multiplication of duals and coefficients from the cost of the new column. Since, in this cutting stock problem, the cost of the pattern is 1 for all patterns, the total multiplication can be subtracted from 1. Also, the constraint of the model can be seen in Equation 4.6. It is the capacity constraint forcing that the sum of the cuts in a roll of stock should not exceed its length. The solution values are the coefficients of the new column to be added to the master model if a solution with a negative objective function is found for the sub-problem. Consequently, the variables A_i must be nonnegative integers. This is a type of knapsack problem.

4.2.3. Modeling with ICRON

Before constructing the algorithm, the data model of the cutting stock problem mentioned previously needs to be adapted to the structure of the Gilmore-Gomory column generation algorithm. As can be seen in Figure 4.7, there are two new classes associated with the problem and needed to be under this model in order to manage the operations and the attributes of the new decision variable within the object world. The attributes of these classes can be also seen in the data model of the problem in Figure 4.8. One is “MP_Column” class which has a generic structure symbolizing one column of the solver matrix with the coefficient values of the associated decision variable in the objective function and the constraints. It is already defined in the system (see section 3.2.8) and can

have objects when the column related nodes under the mathematical programming module are used in an algorithm. By default, an MP_Column object is created without any coefficient value. Thus, the user should assign the coefficient values in anywhere in the algorithm suitably. This is achieved by creating appropriate fields in the data model so that the user can assume them as the object oriented representation of the coefficients. Thus, the user can assign values to these fields and then set the values of these fields to the appropriate attribute of the MP_Column object by using functions defined in the system. Since one column object (decision variable) has only one coefficient value in the objective function at a time, a field for the objective function coefficient is created under the MP_Column class. In this model, “ObjCoeff” refers to this field.

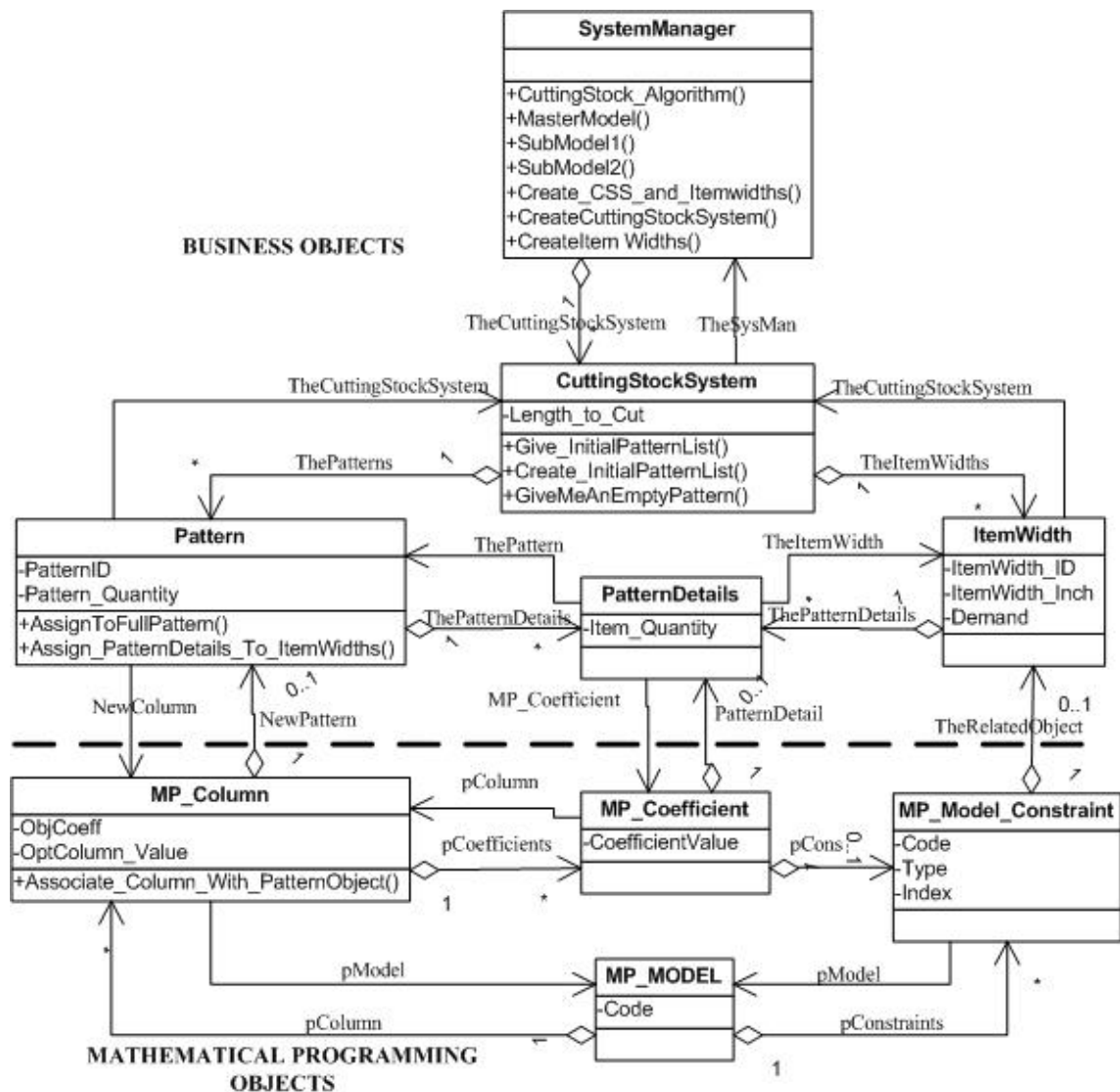


Figure 4.7. The class diagram of the cutting stock using Gilmore-Gomory Algorithm.

There is also an algorithm defined under MP_Column class, “Associate_Column_With_PatternObject”. This algorithm creates a new pattern object since it is the decision variable object in this model, and then assigns the constraint coefficients of the column to the appropriate fields of the pattern details of this pattern object. Thus, the created column object will have its business-object representation in the model so that the user can easily translate the solution in terms of the real objects.

Beside the coefficient value of the decision variable in the objective function, the user might want to assign the coefficient values in the constraints during run time. Since the number of the coefficients depends on the number of the constraints and each coefficient is directly and strongly associated with a constraint, a new class “MP_Coefficient” is created under the model. Then, each column object has its MP_Coefficient objects which are as many as the number of the constraints in the master model and set automatically by the system when the column is created. MP_Coefficient objects are created with the information of the constraint by default and its “CoefficientValue” attribute holds the user-defined value of the coefficient of the decision variable in the constraint which the MP_Coefficient object is associated.

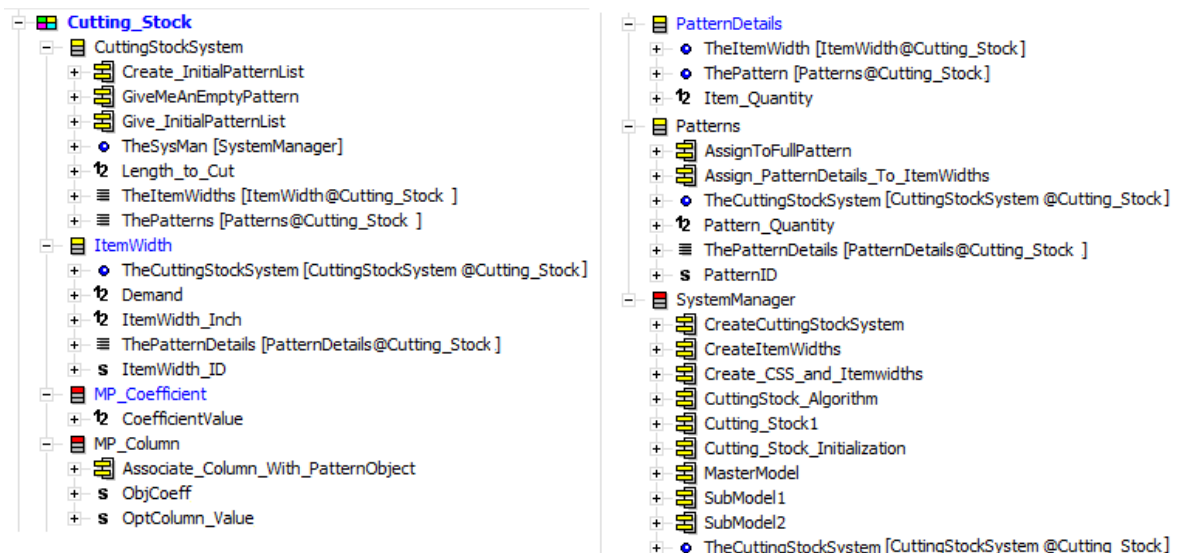


Figure 4.8. Data model.

The change from the model of the first example is not only these two new classes. The “Cutting_Stock” model has also some additional algorithm models created under some

of the classes. For example, “CuttingStockSystem” class has three algorithm models: “Give_InitialPatternList”, “Create_InitialPatternList”, and “GiveMeAnEmptyPattern”. There are two algorithm models under the “Pattern” class, too: “AssignToFullPattern” and “Assign_PatternDetails_To_ItemWidths”. Only one algorithm is associated with MP_Column class: “Associate_Column_With_PatternObject”. Lastly, “SystemManager” class has six new algorithms: “CuttingStock_Algorithm”, “MasterModel”, “SubModel1”, “SubModel2”, “Create_CSS_and_Itemwidths”, “CreateCuttingStockSystem”, “CreateItem Widths”. These algorithm models will be explained later.

Recall that the Gilmore-Gomory algorithm proposes four steps to model a cutting stock problem. First, a master model is constructed with the initial pattern list. Then, a submodel using the duals of the master model is prepared to find one more decision variable/column to the master model to reduce its objective function. In the third step, if the column is found, it is added to the master model. Then, the iteration between master model and submodel continues in this manner until there is no beneficial column. At the last step, the master model is solved as a MIP, and then the solution of the problem concludes. The complete algorithm where all of these steps are gathered into is modeled in the “CuttingStock_ Algorithm” which can be seen in Figure 4.9. The flow of this algorithm and each step of the algorithm are explained in detail below.

As the first step of the algorithm, a cutting stock problem is solved as an LP by setting the initial pattern list as the decision variable objects with real types. Since, column generation algorithms need great modeling effort with so many operations that make controlling more difficult, grouping of some operations in the model appropriately is an effective way to ease this modeling process rather than modeling all operations within a single algorithm. In this object oriented modeling environment, this is achieved by using separate algorithms as a component of another algorithm. As we explain the details of the algorithm later, the “MasterModel” fulfills the operations in the first step of the cutting stock problem where it is basically used as a component which takes “SystemManager” object as an input object. Its outputs, “Object” and “MP_MODEL”, carry the manipulated “SystemManager” object and the model object, which is created for the master model and holds the information about its solver and all modeling objects in the master model, respectively. Unless this model object is lost in runtime, it will be possible to

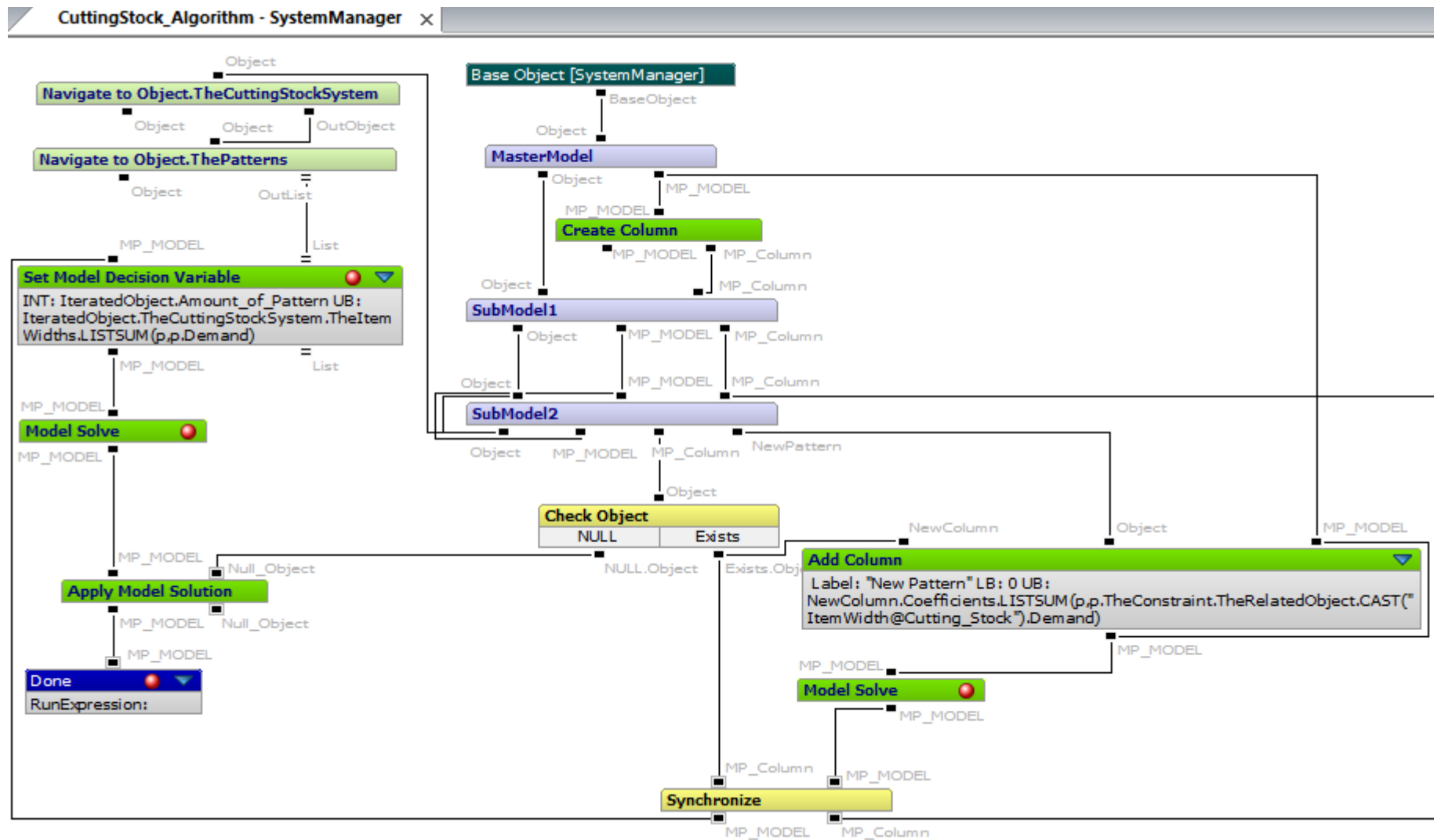


Figure 4.9. Mathematical model of cutting stock problem using the Gilmore-Gomory Algorithm.

manipulate and update the last information of its objects and its associated solver's output.

At the second step of the cutting stock problem solved by using column generation, an algorithm is constructed so as to find a decision variable that will improve the objective function of the master model. The submodel algorithm mentioned above performs these optimization operations together with its two parts: SubModel1 and SubModel2. Likewise the MasterModel, these parts of the submodel algorithm are used as components of the cutting stock problem. In order to run the submodel algorithm, a mechanism is needed to carry the duals of the constraints in the “current” model object of the master model to the submodel. This is achieved with a column object. An MP_Column object created by “Create Column” node is automatically in association with the model object given as the input object, and its empty MP_Coefficient objects is automatically associated with the constraints of the current model object. Then, the column object is linked to the SubModel1 component. As mentioned later, this first part of the submodel creates another model object for the submodel, the decision variables, and the constraints of the submodel. Then, its outputs hold the system manager object in the “Object” link point, the model object of the submodel in the “MP_MODEL” link point, and the column object in the “MP_Column” link point. All these link points are tied with the corresponding input link points of the “SubModel2”. The separation of the submodel has no significance in the first run of the submodel. It will be critical in the next iterations as explained later. Next, the “SubModel2” is called to create the objective function of the submodel and to solve it by using the information in the model object of the submodel. The output of the submodel is the system manager object, the modified model object of the submodel, the modified column object, and the pattern object which is created by using the coefficient information of the column object.

In the third step of the algorithm, it is checked whether a valid column is generated by the submodel or not, and then if found, the column is added to the master model. This process is modeled in the algorithm seen in Figure 4.9 via “Check Object” node and “Add Column” node. The “Check Object” node partitions the flow of the algorithm into two separate ways: If the column object taken from the submodel exists with valid coefficient values, then this column is added. Otherwise, the algorithm jumps to the last step. For the column addition, “Add Column” node is used. The column object is given as input to the

“NewColumn” link point, the decision variable object of the master model, which is “pattern” object and created newly after the column is found, is linked to the “Object” point, and the model object of the master model is tied to the “MP_MODEL” point. Then, the form of the node is filled as in the Figure 4.10. By using this node, the “Pattern_Quantity” attribute of the pattern object, which is recently created, is set as the new decision variable of the master model. This can be seen in the “Decision Variables” list of the model. In addition, the new column is added to the last output of the solver so that the solver output is only updated instead of being restarted.

Set column label, lb and ub	
Decision Variable	Object.Amount_of_Pattern
Column Label	"New Pattern"
Lower Bound	0
Upper Bound	NewColumn.Coefficients.LISTSUM(p,p.TheConstraint.TheRelatedObject.CAST("ItemWidth@Cutting_Stock").Demand)
OK Cancel	

Figure 4.10. Add Column node in the CuttingStock_Algorithm.

Since the output of the submodel is adapted to the master model by adding the column, it is possible to solve the current master model. This starts the next iteration starting with the first step. Here the model object, fired from the MP_MODEL fixed output link point of the “Add Column” node, is the object of the master model. Since it holds both the updated modeling objects and the updated solver data, there is no need to recreate the same model, objective function, constraints, and decision variables of the master model. That is, the current model object of the master model is ready to be solved with “Model Solve” node. Then the master model is solved by using this node. When the algorithm is run until the “Synchronize” node, the column object and the master model object are accessible on the mode. After the master model is solved in the first iteration, next process was to create a column. However, since it is already accessible at this point and its coefficient values were stored in a pattern object appropriately, this column object can be used again in the next iteration. Then, in the second step, the decision variables and the constraint of the subproblem was created. But, they still have the same values and

attributes in the next iteration and so there is also no need to recreate them. Then, in the SubModel2, the objective function of the submodel was created and then the model was solved. However, this part of the model is different in this iteration because the multiplier in the objective function, the dual value, changed with the solution of the master model. In order to continue with updating the objective function of the submodel in this iteration and resolving the submodel, a link is constructed between the “Synchronize” node and the “SubModel2” algorithm node. It is important in the objective function that the duals are accessible via the “TheConstraint” object of the each MP_Coefficient object which is in the coefficient list of the column object. This means that the column object on the “Synchronize” node allows reaching the new duals also. Thus, MP_Column output link point of this node is tied with the MP_Column input link of the “SubModel2”. In order to run the SubModel2 again, its other input link points should also be triggered at this point. Thus, the other outputs are tied with the corresponding input link points of SubModel2. Then, the submodel is solved with new duals by using the current model object of the submodel, and then it may further propose a new column object again. Here it is important that, like the solver data of the master model is updated whenever a column is added, the solver data of the submodel is also updated when the duals of the objective function is changed. This is the functionality of the system so as to decrease the solution time by eliminating the repetitions.

The loop between the master model and the submodel continues until a valid column is not obtained from the solution of the submodel. If so, the algorithm passes to the last step: the master model is solved as a MIP. Since the decision about the column object is made in the “Check Object” node, its NULL option should be used to cut the loop, and to direct the algorithm to run the master model with integer decision variables. Since there is no need to use this null column object in the rest of the algorithm, the “Null.Object” of the “Check Object” node can be linked to anywhere except the loop. In this example, it is linked to the “Apply Model Solution” node on the left part of the window. When the algorithm runs the Check Object node and the loop is cut due to the null column object, the next node to be run will be “Navigate to Object.TheCuttingStockSystem” which becomes the candidate path after each running of the “SubModel2” algorithm node due to the link between these two nodes. Then, in order to change the types of the decision variables in the master model from real to integer, all of the pattern objects are obtained via the navigation

nodes in a similar way in the master model. By giving this pattern list and the current model object to the “Set Model Decision Variable” node, the same attribute is defined as the decision variable and the type is selected as integer. When the node is run, the types of the decision variables are updated both in the model and in the solver without setting new decision variables since they have been already created. Then, the model is ready to be solved with integer decision variables. “Model Solve” is called for the last time, and the solution is applied to the decision variable objects.

As defined previously in column generation method, the problem is modeled as one master model and several appropriate submodels. The “MasterModel” under the “SystemManager” class is, as its name implies, the master model of this problem. The algorithm can be seen in Figure 4.11. It is almost the same with the general formulation of the problem modeled in the first example. The difference is related with the initialization of the problem, the type of the decision variables, and defining a “Related Object” to the constraints. The initialization of the cutting stock system and of items is similar with that of the first example. However, the patterns which are the decision variable objects in this problem are initialized differently. According to the algorithm, only the initial set of decision variables should be considered in the first iteration of the model. Thus, an initial pattern set is constructed based on the formulation in Figure 4.6. Since we have four constraints created for four items having different lengths, the initial patterns set have four patterns. As can be seen in Table 4.6, each pattern includes the information of maximum quantity of cuttings of an item from a roll of stock. The details of the creation of patterns are not discussed within the scope of this work.

Table 4.6. The Initial set of the cutting stock example.

	Width1	Width2	Width3	Width4
Pattern1	Floor(100/45)=2			
Pattern2		Floor(100/36)=2		
Pattern3			Floor(100/31)=3	
Pattern4				Floor(100/14)=7

After the initialization of cutting stock system, items, patterns and their details, a model object is created with the information of the solver which is defined by the user when the algorithm is created. Then, the objective function of minimizing the total number of used patterns is created. Then, the decision variables are defined similarly with the first

example. Note that, the decision variable, the quantity of pattern, is not integer in this model. As the Gilmore-Gomory algorithm dictates, they should be “real” until there is not a possibility to obtain a new decision variable that will improve the objective function. That is, the master model in the cutting stock problem using the Gilmore-Gomory algorithm is not solved as a MIP but as an LP in order to be able to obtain a dual optimal solution.

The other change is to define a “Related Object”. While defining the constraints, “ItemWidth” object is defined in the “Related Object” field in the dialog box of the node. This object is selected because each constraint in this example is characterized by the ItemWidth object, and this object may be needed later to use in any other model. By defining ItemWidth object as related object, it will be possible to reach it through the constraints of this model object because each constraint will hold the information of its associated ItemWidth object. Then, the rest of the algorithm is modeled like in the first example.

The decomposition of the models into several algorithms and their usability within each other like a node provide the user to follow up the process easily and to reuse the algorithm in other algorithms if needed. For example, this master model, in fact, has more algorithms which are not seen in Figure 4.11. The “Create_CSS_and_Itemwidths” algorithm seen in the first phase of the initialization has “CreateCuttingStockSystem” and “CreateItemWidths” algorithms inside. That is, cutting stock system and the items are initialized in different algorithms and then gathered in one another algorithm. The other algorithm on the figure is “Give_Initial_PatternList”. It has two more algorithms. First one “Create_Initial_PatternList” creates the initial pattern list by gathering the pattern objects which are created in a different algorithm: “GiveMeAnEmptyPattern”. After that, the PatternDetail objects related with these patterns are associated with the item objects in the algorithm “Assign_PatternDetails_To_ItemWidths”.

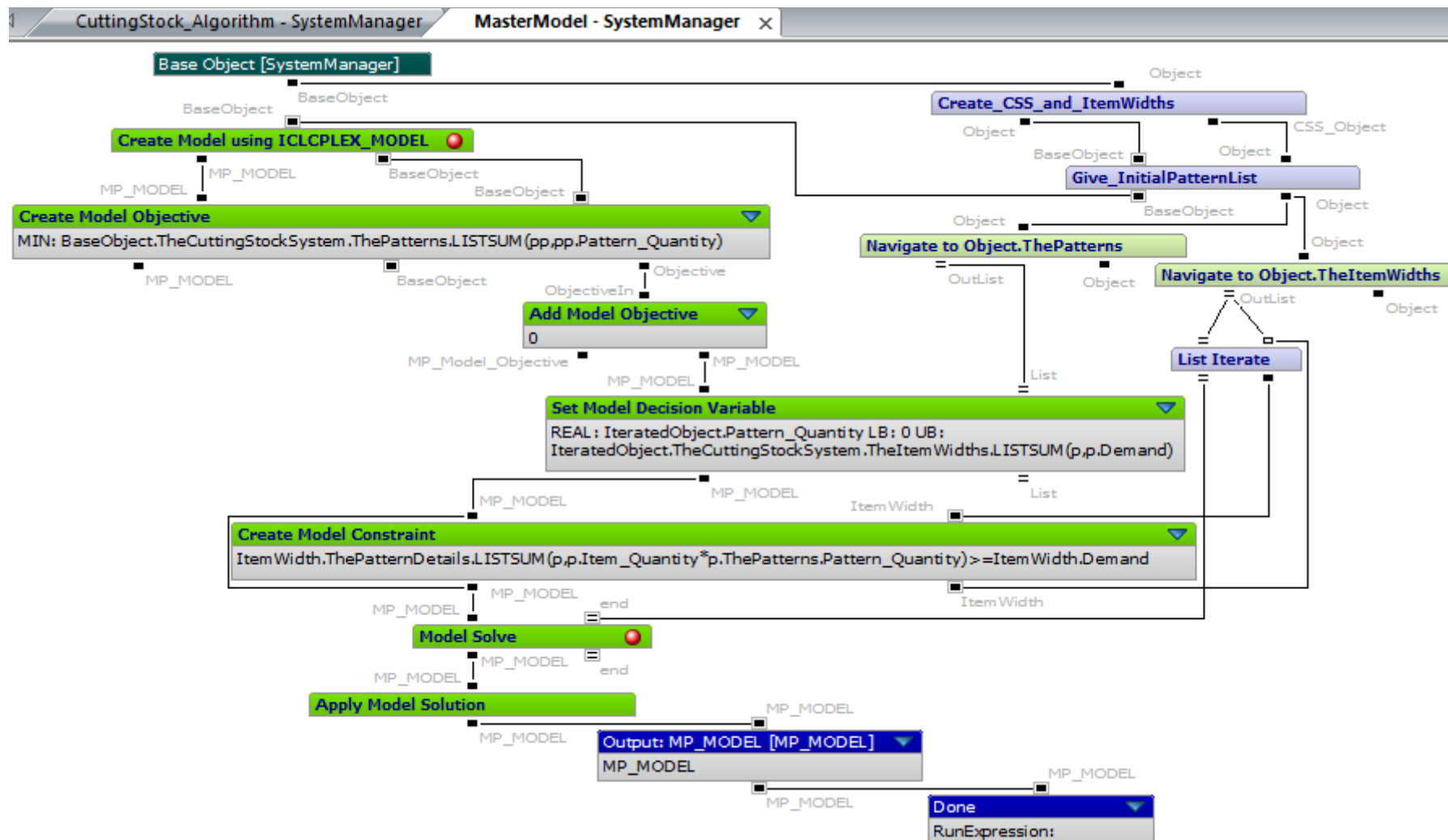


Figure 4.11. Master model.

As can be seen, the most of algorithms created for this cutting stock problem are used in the modeling of the master problem. Then, the column generation technique suggests that there should be also a submodel which will take the dual values of the constraints from the master model and then in return will give necessary information of the decision variable of the master model if it has an optimum solution. Before going further on details about the submodel algorithm in this cutting stock problem, it will facilitate a clear understanding to describe the relationship between decision variable, pattern objects, and column objects.

As Figure 4.12 shows, while pattern and pattern details objects are used in the master model, column and coefficient objects are used in the sub model. As we know that the decision variable objects of the master model are pattern objects, and the initial decision variable set is constructed with the initial patterns mentioned above. Then, the submodel uses the dual information of the master model in its objective function and optimizes its mathematical model as explained later. Generally, it is expected sub model to offer a new decision variable to the master model. It corresponds to a new column in the solver. Then, the MP_Column object, which carries the name of the decision variable in the solver, symbolizes the new decision variable object in the master model. Like the decision variable object, pattern, having pattern detail objects to hold the relative coefficient values in the constraints, MP_Column object is associated with MP_Coefficient objects for the same purpose. If the sub model provides a valid column, it should be added to the previous decision variable list, to the solver as the new column and to the pattern list of the master model in the appropriate formats. Then, the process repeats the same procedures in the master model and the submodel for the next iteration.

The submodel algorithm in the cutting stock problem using the Gilmore-Gomory algorithm is the example of the algorithms grouped into separate algorithms mentioned above. It can be divided into two separate algorithms one of which includes the components affected with the iteration, and the other includes unaffected ones.

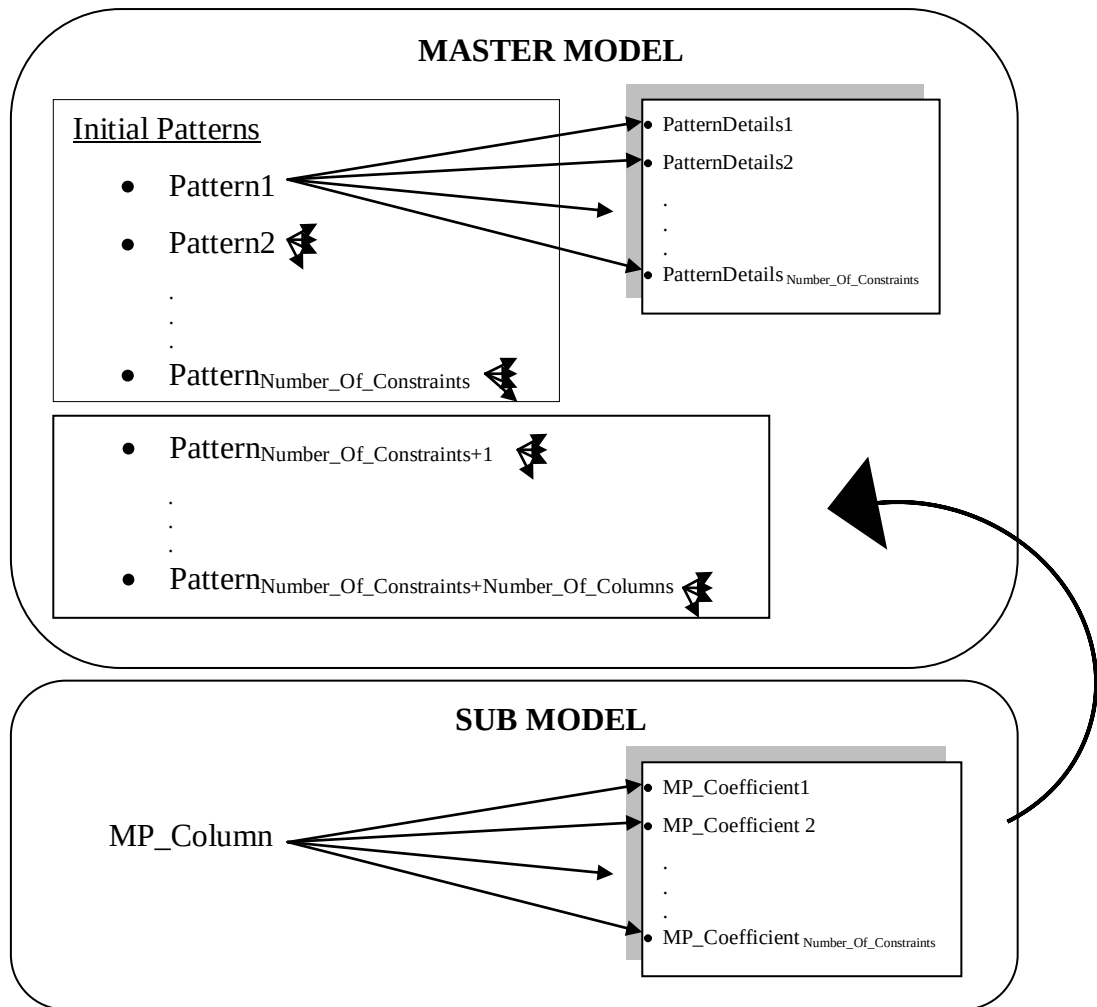


Figure 4.12. The relation between Pattern and MP_Column objects and between PatternDetail and MP_Coefficient objects.

The “Sub Model1” algorithm seen in Figure 4.13 includes the components which only need to be set-up once. The algorithm starts with the creation of its own model object as associated with the solver defined by the user. After the model object of the sub model, the decision variables and the constraints are created as associated with it. The decision variable of the sub model is the “CoefficientValue” attribute of MP_Coefficient objects which are automatically created when the column object is built in runtime. The creation of the column object will be explained later, but here it is important that a predefined column object is given to this algorithm as an “Input object”. Then, the constraint of the submodel is built as the capacity constraint seen in Equation 4.7. The LHS value of the constraint is the length of the stock. The RHS expression simply sums up all multiplications of

coefficient values and the widths of the items. In order to make the multiplication correctly, it is needed to know the relationship between the MP_Coefficient objects and item objects. When a column is created, each MP_Coefficient object is constructed as associated automatically with a constraint object of a model to which the column will be added later. In this equation, “TheConstraint” refers to the constraint object associated with the iterated MP_Coefficient. Since the column will be added to the master model, these constraints denote the demand constraints of the master model. As we remember, each constraint in the master model has a related object which is defined as “ItemWidth” object by the user. In order to access the attributes of the related object, “CAST” function is used as in the equation. Then, “CoefficientValue” is multiplied with “ItemWidth_Inch” attribute which holds the value of the items. Since this equation is generic and the length of the stock and the width of the items do not change throughout the solution of a cutting stock problem, it is not needed to define the constraint over and over again in each call of the submodel. This is valid in setting the decision variables, too.

$$\text{MP_Coefficients.LISTSUM}(p, p.\text{CoefficientValue} * p.\text{TheConstraint}.\text{TheRelatedObject}.\text{CAST}(\text{"ItemWidth@Cutting_Stock"}).\text{ItemWidth_Inch}) \leq \text{CSSObject.Length_to_Cut} \quad (4.7)$$

At the end of the first part of the sub model, two objects are returned from the algorithm as “Output Object”. These are the column object and the modified model object created for the submodel. They will be used later in the rest of the sub model algorithm.

Since there is no enforced order between the creation of decision variables, constraints, and the objective function, it does not change the solution when the objective function is created later. The second part of the submodel seen in the Figure 4.14 begins with the definition the objective function. By definition, the objective function node needs a model object as the fixed input object. Additionally, as can be seen in Equation 4.8, the objective function navigates over a column object to reach the list of coefficient objects. Therefore, MP_Column and MP_MODEL objects are given to the model as “Input Object”. The objective function of the submodel is to get the most negative reduced cost generally defined as the subtraction of the total multiplications of duals and coefficient values of the new column from the cost of this column. Instead of this type of

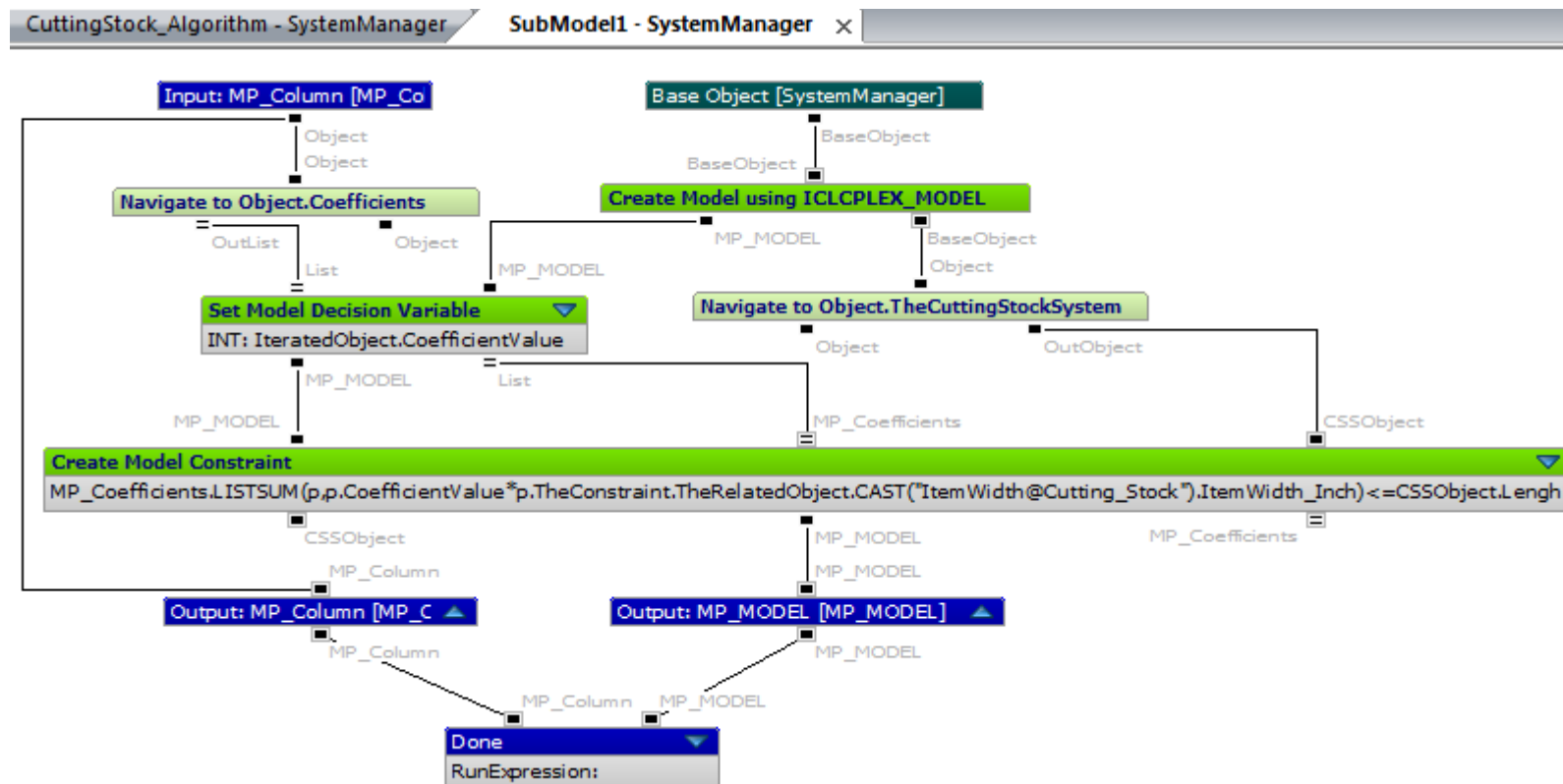


Figure 4.13. SubModel1 algorithm.

formulation, we can maximize the multiplication in the objective function and then check if the optimum objective value is positive or negative. Its negative value will give the minimum reduced cost similar to the original formulation. Then, the objective function in Equation 4.8 multiplies the coefficient value of each MP_Coefficient object with the dual value of the constraint which the coefficient object is associated with. After that, all multiplications are summed up.

$$\text{MP_Column.Coefficients.LISTSUM}(p, p.\text{CoefficientValue} * p.\text{TheConstraint.Dual}) \quad (4.8)$$

After the objective function is defined, the sub model is ready to be solved. The MP_MODEL object in the fixed input link point of the “Model Solve” node is the modified model object of the sub model, and it carries all the required information to solve the sub model. After the model is solved, the objective value of the solution is checked as we mentioned above. If the subtraction of the objective value from 1, which is the cost of the column, is negative, it is the most negative reduced cost and thus this column can improve the objective function of the master model when it is used as a decision variable.

If the sub model finds an optimum solution and offers a valid column object, then the column is prepared to make the necessary modifications on the master model and its solver. Firstly, the solution of the decision variables are applied to the “CoefficientValue” fields of the MP_Coefficient objects by the “Apply Model Solution” node. Then, the user assigns the coefficient value of the column in the objective function, which is the cost of the column, by the “Assign” node. At this step, being assigned by the optimization of a model and by the user, these coefficient values in the constraints and in the objective function are hold only in the “CoefficientValue” and “ObjCoeff” fields which are created within the GSAMS environment. In fact, these assignments do not affect the appropriate coefficient fields of the column object within the system. However, by using these fields created in the GSAMS environment as the parameter of some built-in functions, their values can be applied to the appropriate fields of the column object in the system. The “MPCoeff_SetCoefficient (coefficient value)” and “MPCol_SetObjectiveCoefficient (coefficient value)” functions are used for this purpose by using “Evaluate” nodes. These assignments to the column object in the system are required so as to add the column to the master model as the new decision variable which will be explained later.

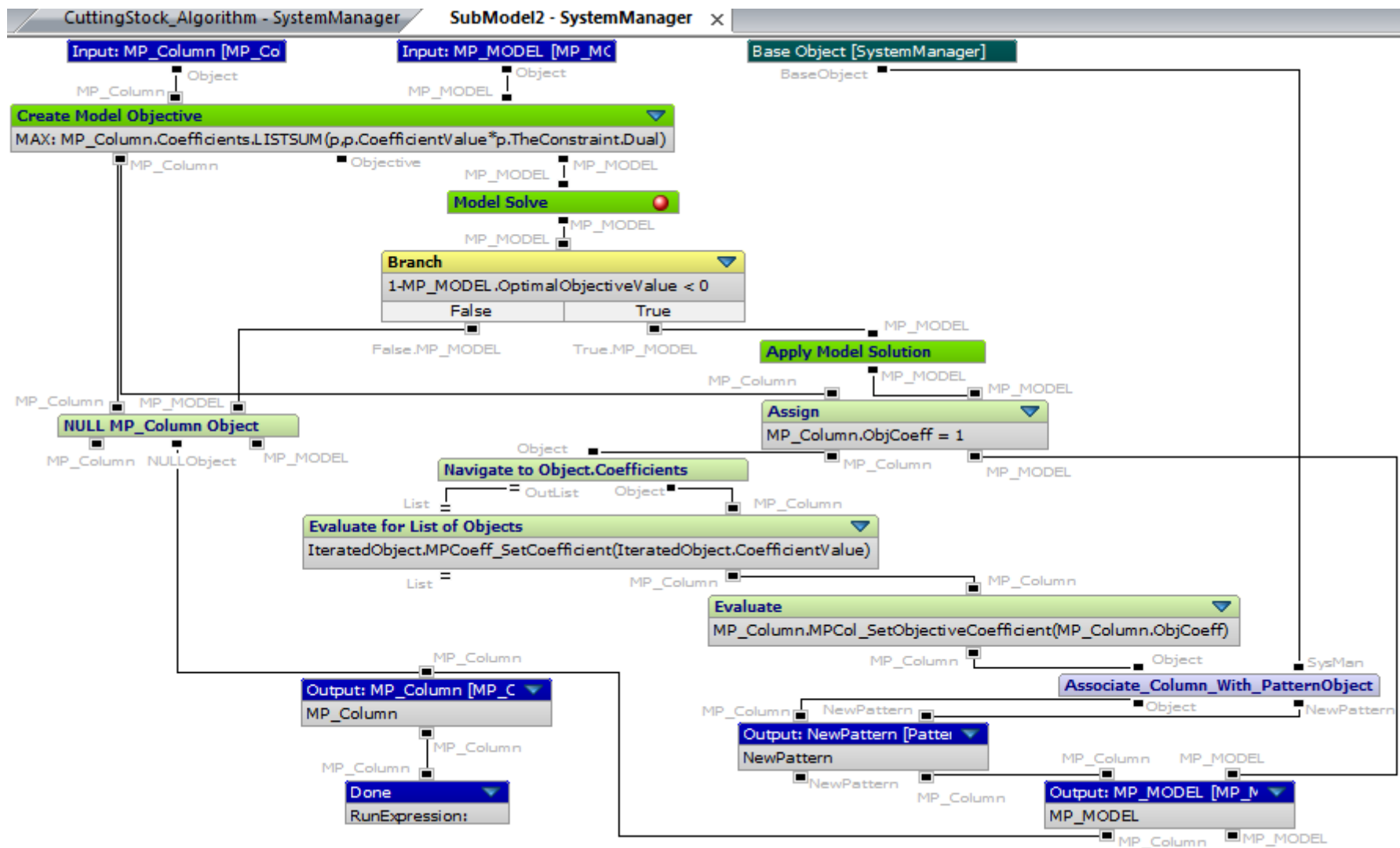


Figure 4.14. SubModel2 algorithm.

Then, there is an algorithm “Associate_Column_With_PatternObject” which is used as a component in this sub model algorithm. The algorithm includes almost the same procedures with definition of the initial pattern list in the master model. An empty pattern object is created as associated with empty pattern detail objects. Then, their appropriate relations with the cutting stock system and the item objects are constructed. As differently, the fields “Item_Quantity” of the pattern detail objects are filled with the constraint coefficient values of the new column object. As mentioned above in the Figure 4.12, the MP_Coefficient objects hold the information for the pattern detail objects of the new pattern object. The assignment of the value is fulfilled by defining two expressions for the field and the value. The LHS of the expression seen in the Equation 4.9 is the appropriate field of the iterated pattern detail object. The RHS is the expression of the value of the corresponding coefficient object. Since the only association between pattern detail objects and the MP_Coefficient objects is the ItemWidth object, the objects which have a relationship with the iterated ItemWidth object are picked from the both lists by the LISTPICK function. At the end, we have a column and new pattern objects ready to be added to the master model. Before that, since each decision variable object in the master model is created with the information of a pattern object, it will be beneficial to assign this new pattern object as the decision variable object of this column object. This is achieved with the function “MPCol_SetDVObject(NewPattern)”. At the end of the algorithm, the modified column, pattern, and the model objects are fired from the algorithm as an “Output Object” so as to be used later in the flow of the Gilmore-Gomory algorithm.

$$\text{EmptyPattern.ThePatternDetails.LISTPICK (p,p.TheItemWidth==ItemWidth).Item_Quantity=MP_Column.Coefficients.LISTPICK(p,p.TheConstraint.TheRelatedObject==ItemWidth).CoefficientValue} \quad (4.9)$$

If an optimum solution is not obtained and thus the coefficients of the column object cannot filled with valid values, a “NULL Object” is created so that the column object fired from this algorithm can be checked as null or not.

4.2.4. Solution of Cutting Stock Problem

While the first cutting stock problem in the previous example has 37 decision variables, the problem in this example started to be solved with four decision variables which were the objects in the initial pattern list. Then, the submodel had a mathematical model seen in Table 4.7. The values in the objective function are the duals of the master model. “CoefficientValue” attribute of the MP_Coefficient is the decision variable. There is a single capacity constraint in which the coefficient values are multiplied with the widths of the items.

Table 4.7. The mathematical model of submodel in the first call.

Type	Z row			
MAX	$0.5 * \text{CoefficientValue}_0 + 0.5 * \text{CoefficientValue}_1 + 0.3333333333333333 * \text{CoefficientValue}_2 + 0.14285714285714 * \text{CoefficientValue}_3 + 0$			
Name		LHS	Type	RHS
Capacity constraint_Expanded		$45 * \text{CoefficientValue}_0 + 36 * \text{CoefficientValue}_1 + 31 * \text{CoefficientValue}_2 + 14 * \text{CoefficientValue}_3 - 100$	$\leq$	0

The optimal values of the submodel in the first call can be seen in Table 4.8. This is the new column to be added to the master model.

Table 4.8. Optimal values of the submodel in the first call.

	Name	Optimum Value
0	MP_Coefficient1	0
1	MP_Coefficient2	2
2	MP_Coefficient3	0
3	MP_Coefficient4	2

After the new column is added to the master model, the model is solved again. Then, new dual values are obtained from the solution of the master model. The new mathematical model of the submodel can be seen in Table 4.9. The only change from the first model is the objective function because of the dual values.

Table 4.9. The mathematical model of submodel in the second call.

Type	Z row		
MAX	0.5 * CoefficientValue_0 + 0.5 * CoefficientValue_1 + 0.33333333333333 * CoefficientValue_2 + 0		
Name	LHS	Type	RHS
Capacity	45*CoefficientValue_0 +36*CoefficientValue_1+	<=	0
constraint_0_Expanded	31*CoefficientValue_2+14*CoefficientValue_3 - 100		

The optimal values of the submodel in the second call can be seen in Table 4.10. This is the second column to be added to the master model.

Table 4.10. Optimal values of the submodel in the second call.

	Name	Optimum Value
0	MP_Coefficient1	0
1	MP_Coefficient2	1
2	MP_Coefficient3	2
3	MP_Coefficient4	0

After the third call of the submodel, a valid column is not generated, and the master model is solved as a MIP with these six decision variables which are the initial pattern list plus two new patterns. The pattern objects at the end of the model are listed in Table 4.11. The names of these pattern objects are given sequentially without depending on the all possible pattern list demonstrated previously. According to the solutions of the submodel, Pattern5 and Pattern6 have more potential to improve the objective function of the master model relative to the other alternative patterns.

Table 4.11. The pattern list in the last master model.

		Width1	Width2	Width3	Width4
0	Pattern1	2			
1	Pattern2		2		
2	Pattern3			3	
3	Pattern4				7
4	NewPattern0		2		2
5	NewPattern1		1	2	

When the master model is solved with these patterns, a mathematical model is constructed as can be seen in Table 4.12. “Pattern_Quantity” attributes of the patterns are

again the decision variables of the problem. The objective function has only six decision variables in this example.

Table 4.12. Mathematical model of cutting stock model using column generation.

Type	Z row			
MAX	Pattern_Quantity_0+Pattern_Quantity_1+Pattern_Quantity_2+ Pattern_Quantity_3+ Pattern_Quantity_4+Pattern_Quantity_5+0			
Name	LHS	Type	RHS	
demand constraint_0_Expanded	2*Pattern_Quantity_0 - 97	<=	0	
demand constraint_1_Expanded	2*Pattern_Quantity_1 + 2* Pattern_Quantity_4+ Pattern_Quantity_5 - 610	<=	0	
demand constraint_2_Expanded	3*Pattern_Quantity_2+2*Pattern_Quantity_5 - 395	<=	0	
demand constraint_3_Expanded	7*Pattern_Quantity_3+2*Pattern_Quantity_4 - 211	<=	0	

The optimal solutions of the problem can be seen in Table 4.13 in terms of “Pattern_Quantity”. Among six pattern objects, four of them have positive values. The optimum policy is to use Pattern1 for 49 times, Pattern2 for 106 times, NewPattern0 for 100 times, and NewPattern1 for 198 times. When the pattern details are compared with the all possible pattern list in Table 4.2, these patterns correspond to the Pattern1, Pattern10, Pattern12, and Pattern13 objects similarly with the solution of the first model. The total quantity is again 453. The upperbound of the decision variable is again the sum of the demands, 1313.

Table 4.13. Optimal solutions of cutting stock model using column generation.

Optimal Objective Value			
453			
Name	Optimum Value	Lower Bound	Upper Bound
Pattern_Quantity_0	49	0	1313
Pattern_Quantity_1	106	0	1313
Pattern_Quantity_4	100	0	1313
Pattern_Quantity_5	198	0	1313

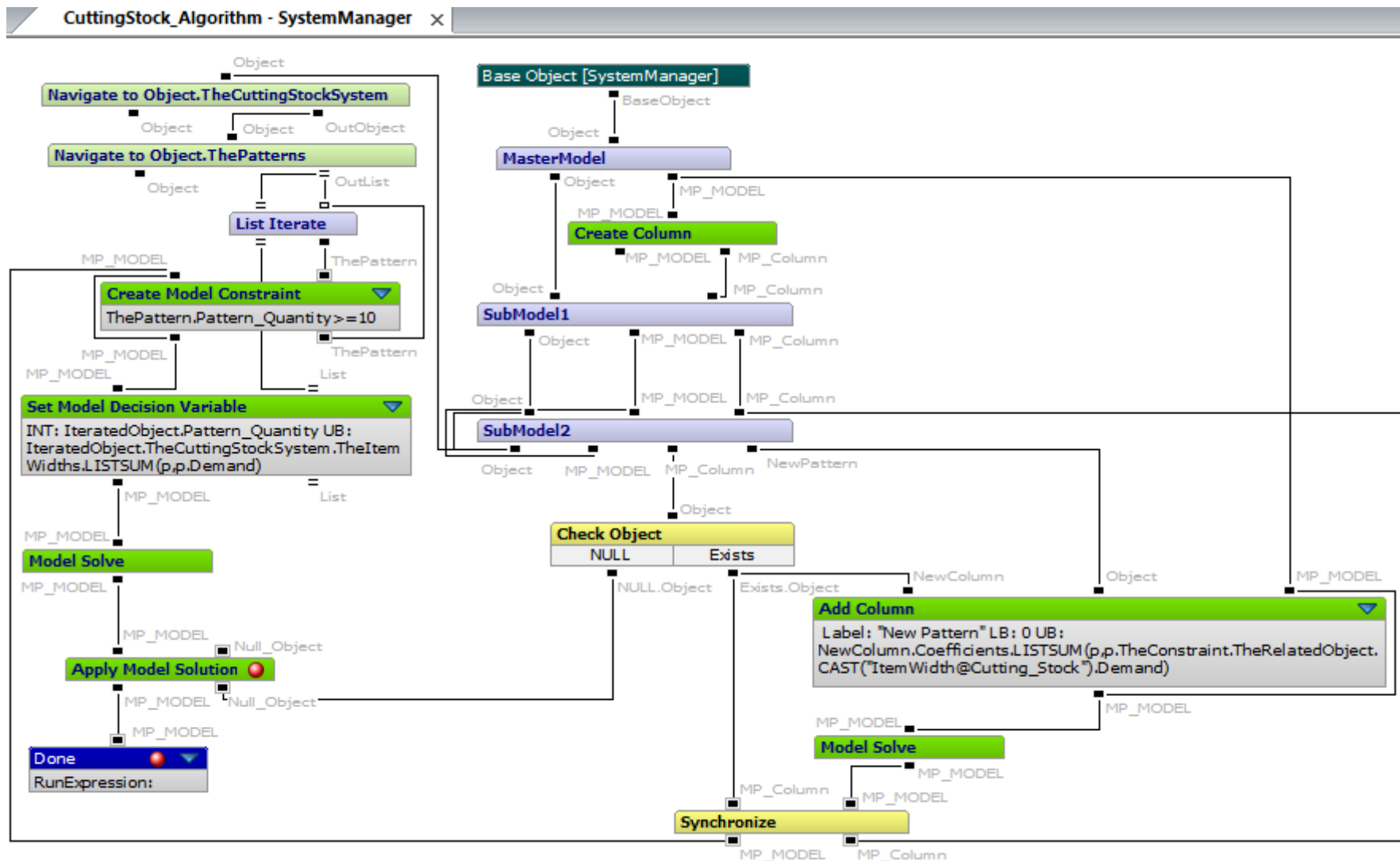


Figure 4.15. The cutting stock algorithm with additional constraints.

4.2.5. Adding a New Constraint to the Cutting Stock Problem

The previous example demonstrated how to add a new decision variable to a model, how to update the objective function of a model, and how to change the type and bounds of a decision variable of a model where the model objects have been constructed previously. Another functionality of the system is also to be able to add new constraints to a constructed model. The “CuttingStock_Algorithm” with new added constraints can be seen in Figure 4.15.

Assume that the quantity of the each pattern must be greater than or equal to 10. In fact, the most efficient way of forcing the lower bounds is to assign the value to the appropriate field in the form of “Set Model Decision Variable” node. However, we will force it by a constraint in this example. The “Create Model Constraint” node is used in a regular way. The expression of the new constraint is: “ThePattern.Pattern_Quantity $\geq$ 10”. After the column generation is finished, the quantities of the patterns are forced not to be less than 10.

When a constraint is added after the model is constructed, the necessary expansion, simplification, and the other procedures are implemented for only this constraint so that its values can be added to the last output of the solver. Since the solver is updated, there is no need to start the solver from scratch.

The optimal solutions of the cutting stock problem with these new added constraints can be seen in Table 4.14. The patterns which are not used in the optimum solution of the previous example, Pattern3 and Pattern4, have the values at their lower bounds. Thus, the other usage of the other patterns changed accordingly. The new optimal value of the objective is 466.

Table 4.14. Optimal values of the cutting stock problem with new added constraints.

Optimal Objective Value			
466			
Name	Optimum Value	Lower Bound	Upper Bound
Pattern_Quantity_0	49	0	1313
Pattern_Quantity_1	143	0	1313
Pattern_Quantity_2	10	0	1313
Pattern_Quantity_3	10	0	1313
Pattern_Quantity_4	71	0	1313
Pattern_Quantity_5	183	0	1313

5. CONCLUSION

In this study, an object oriented mathematical modeling tool, ICRON, is enhanced so that large scale optimization problems can be solved more efficiently by considering the modeling effort of the user and the solution time of the system. The importance of the large-scale optimization problems to represent the real life problems is emphasized. Then, the decomposition techniques in theory to solve this type of optimization problems are represented. Both the capabilities and deficiencies of the traditional and state-of-the-art modeling systems during modeling and solving large scale optimization problems and implementing several decomposition techniques and heuristics are discussed briefly. The significance of the tightness of solver and the modeler is discussed. Some of the modeling tools are evaluated about their tightness. This is followed by the proposition of a new object oriented mathematical modeling tool, ICRON. It is placed as the modeling tool having the tightest relationship between solver and the modeler. The new system, which requires revision of the existing system within the scope of this study to construct a modeling environment in which decomposition techniques can be used easier and solver performances are higher, is introduced. The details of the implementation and how the modeling of new system is achieved are provided. At the end, two case studies and one example as the extension of the second case are presented in ICRON. The models of the examples are constructed and expressed step by step. Thus, the proposed modeling system is demonstrated explicitly.

This new object oriented mathematical programming tool enables users to communicate simply with the system and to construct large scale mathematical programming algorithms easily. Both the object oriented structure of the system and the special functionalities developed to implement several decomposition techniques facilitate the modeling process of algorithms. Since the user deals only with objects, there is no need to create and follow sets, indices etc. which are commonly used in other optimization tools. If the user is familiar with the decomposition concepts such as column and row generation, it will be straightforward to adopt and implement these concepts.

The proposed optimization tool has a tight modeler and solver relationship. Once a model is solved, user can modify and update the last output of the selected solver in

anywhere of the algorithm before the solution terminates. Since any interference to the solver results in an update of the solver output, there is no need to restart the solver from scratch and redo the previous iterations. Hence, the new approach results in significant time saving during execution of the whole algorithm.

APPENDIX

ICRON provides an environment for visual algorithm modeling of mathematical programming problems. It allows users to construct algorithms or customize previously constructed algorithms with respect to the problem needs without coding at all. The system has an object oriented architecture where classes and relationships are constructed and hierarchically organized.

Having a powerful GUI, ICRON provides several explorers which are categorized according to special purposes. User selects any explorer from the list in “Explorer Window” seen in Figure A.1 which pops up via “Open Explorer” command. Then, the selected explorer window, with its information in the form of tree, is placed on the modeling environment so that the user can access easily its components during modeling of algorithms.

An important subset of these explorers can be seen in Figure A.2. The right-upper part of the window is the place where algorithms are modeled in GSAMS environment. The left and bottom part, on the other hand, shows the selected explorers; “Console”, “Models”, “Advanced Nodes”, “Basic Nodes”, “Debug-Link Points”, “MP Messages” and “Object”. These explorers and GSAMS are explained in detail below. Beside the explorers explained above, there are different explorers used for messages purposes in the system, or general purpose manipulations such as task, thread, XML file manipulations, or schedule related explorers such as Jobs Chart, and SF Control. User selects appropriate explorers to perform different functions over modeling environment. The object oriented architecture of the system allows user to visually keep track of any application and to interact with it in GUI via explorers without requiring coding.

On the upper left of the window, “Console” explorer displays the systems that are currently constructed with built-in algorithms in ICRON for a specific application. These algorithms are also constructed by using class definitions in the data model of the system. These classes and algorithms within the system can be changed and extended anytime. If the system has some subsystems, they are listed hierarchically in the tree structure so that user can easily reach each one via GUI to make manipulations. For this particular cutting

stock example, ICRON has only one system, but several built-in class definitions and algorithms. The construction of classes may involve database operations. Usually operating on a database, ICRON has also generic routines for data processing from different data sources

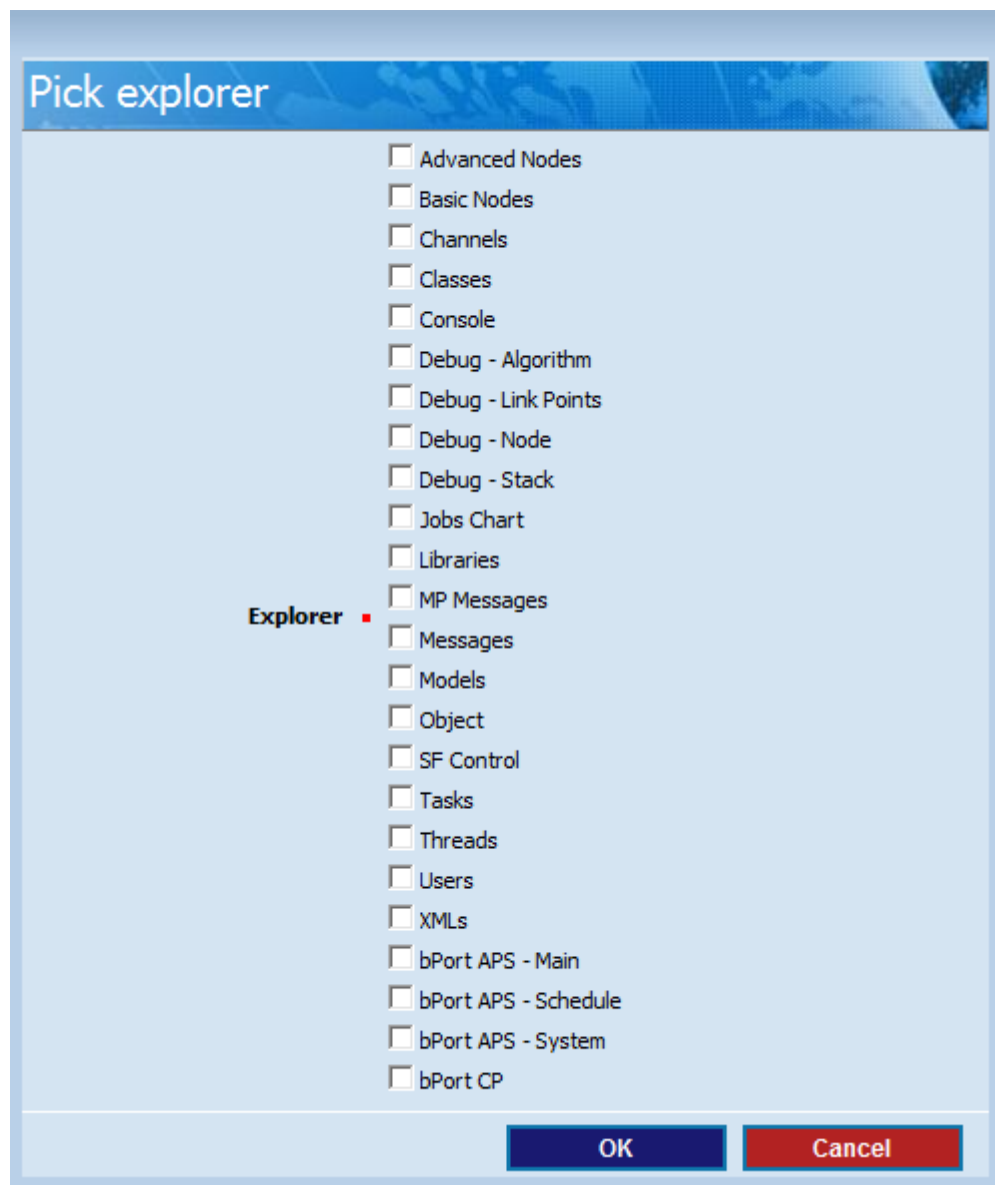


Figure A.1 Explorer window.

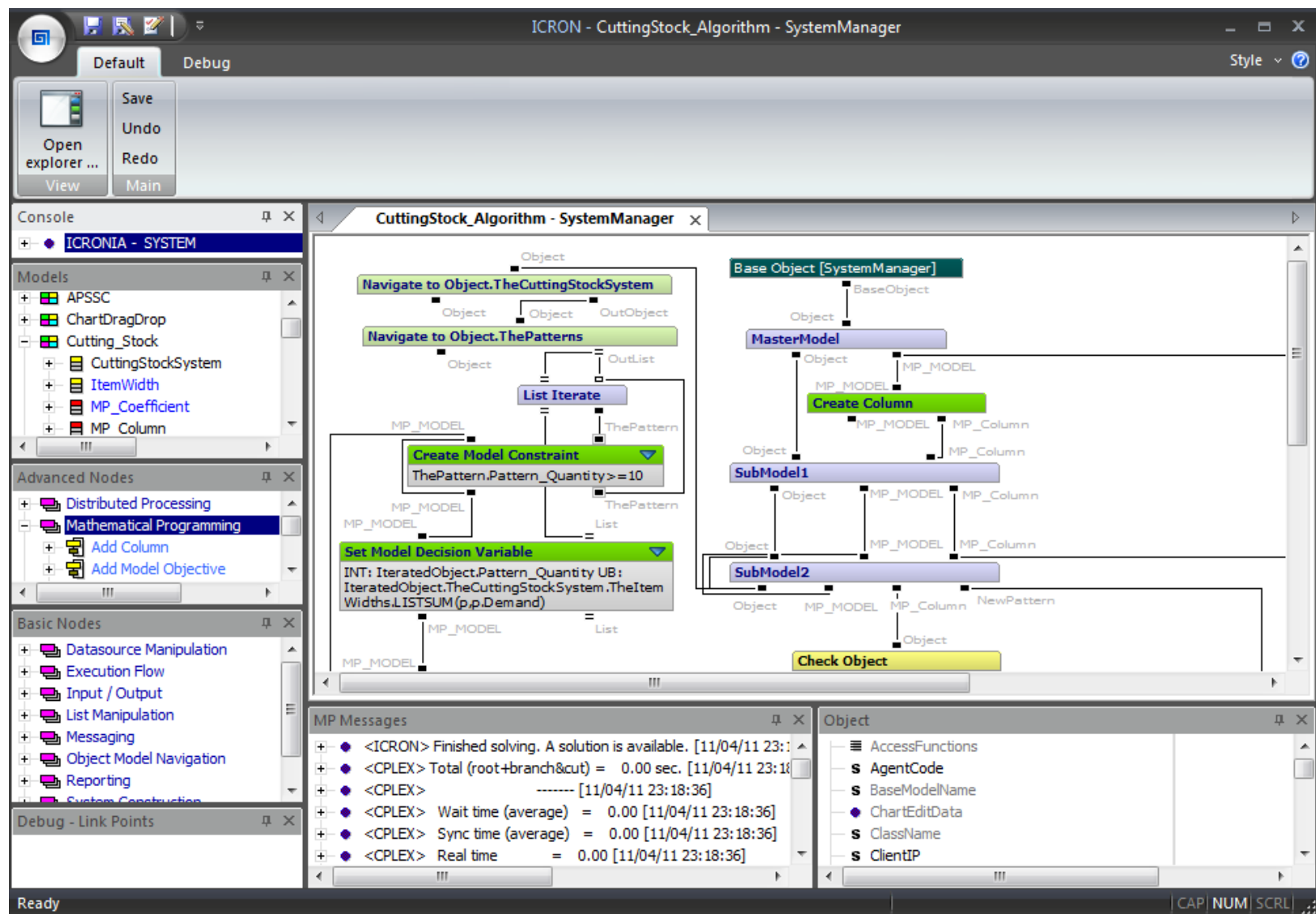


Figure A.2. ICRON environment.

Below “Console”, there is “Models” explorer displaying the constructed data models and allowing users to construct new models and to make modifications on the existing ones. Like the construction of a system, the algorithms in ICRON are also built with object oriented data models. Each model has classes and their relationships. The classes with their attributes, functions and algorithms are listed hierarchically under the model. And in GSAMS, each algorithm is constructed as associated with a class and then it can be used as a method of this class. Any of them can be modified by using pop-up menus with different functionalities which appear after right-click over a class, attribute, function, etc. Then, the created algorithmic models can be executed in the system on appropriate data to manipulate objects of the related classes. Thus, the user can construct algorithm models using GSAMS environment without any programming knowledge. In addition, it is possible to run models for different data in appropriate format. This is the result of data-model independence and object oriented system.

While the model is run in debug mode, “Debug-Link Points” explorer displays dynamically the list of link points of nodes which the debugger triggers currently. Next to the name of each link point in the explorer, the number of input objects is also displayed within parentheses. It is zero when the link point fires. The input objects are listed under the name of the link point. This explorer is used to check the flying objects.

On the bottom of the window, there are “MP Messages” and “Object” explorers. “MP Messages” shows the details of messages during the solution of a mathematical programming model with the information of sender, the message content, and the time of the message. “Object” explorer is used to observe the current information of any object that can be dragged from the object list under any explorer and dropped to the “Object” explorer.

ICRON allows users to debug the execution of the algorithm and thus visually follow step by step running of each node. While it is in debugging mode, if the user wants to see the current details of an object, “Debug-Link Points” explorer can be used so that it is available to obtain information about the currently running node and about objects in the input link points of the node. The object of user’s interest can be dragged from the object list under “Debug-Link Points” explorer and dropped to the “Object” explorer. Thus the

current values of the attributes of this object will be displayed. It helps to observe the changes within the objects until the complete solution is hold.

For the construction of an algorithm, user should firstly define a data model of the problem providing class definitions. The classes used in the algorithm may be defined by the user or among the built-in classes in ICRON. All the definitions of models, classes, and algorithms are managed over the “Models” explorer explained above. By clicking on any model listed in the explorer, user gives the command of creating a new model. Then, by clicking on the model, classes of this model can be created. An algorithm, which has to be associated with a class, is also created with the command given by clicking on the related class. After this command, an algorithm development window pops up automatically and an algorithm is the data model is added under this class. The algorithm development window has a “Base Object [*Class Name*]” node as default. This node implies that this algorithm will be run via the object of the class which is written in brackets and under which the algorithm is created. Afterward, algorithm is constructed graphically using nodes and links.

In ICRON, nodes and links are the visual constructs used for modeling of mathematical programming algorithms within the GSAMS environment. Nodes are rectangular blocks having colors which are sometimes different for a particular set of nodes. The running structure of each node is distinctive with different input-output link points, expression dialog boxes and a mechanism to fulfill a special task. Links represent object flows between nodes. By linking the output link points to input link points appropriately, user can carry objects between several nodes over the algorithm and thus manipulate these objects as desired. Some of the nodes have built-in input and/or output link points called as fixed input/output link points. These link points require the user to link an object or a list of objects to them. While an input link point requiring an object allows only taking on object to this point, an input link point requiring a list of objects permits to tie an object or a list of objects. The other type of link points is user-defined link points. Additional link points can be created by making a link between an output link point of a node and another node via mouse click over the node. Together with the new input link point, an output link points corresponding to this input link point is also added to the node. These are called as dynamic input/output link points.

User can reach nodes via the explorers which group nodes depending on their purposes. The two main ones are “Basic Nodes” and “Advanced Nodes” explorers. Basic nodes are used for general purposes in modeling as related with the manipulation of datasource, management of object flows between nodes, maintenance of input and output of the algorithm, manipulation of a list of objects, messaging, reporting, assignment of values, evaluation of functions, navigation to object(s), XML operations, and construction of a system. Advanced nodes are, on the other hand, used to construct and modify mathematical programming algorithms in GSAMS. They are related with more specific purposes such as distributed processing, rule engine, scheduling, state chart, and mathematical programming, which is our focus point in this study. User drags the node under any group of nodes according to the purpose, and then drops to the algorithm model seen on the right upper part of Figure A.2. Afterwards, user links these nodes by using their link points appropriately.

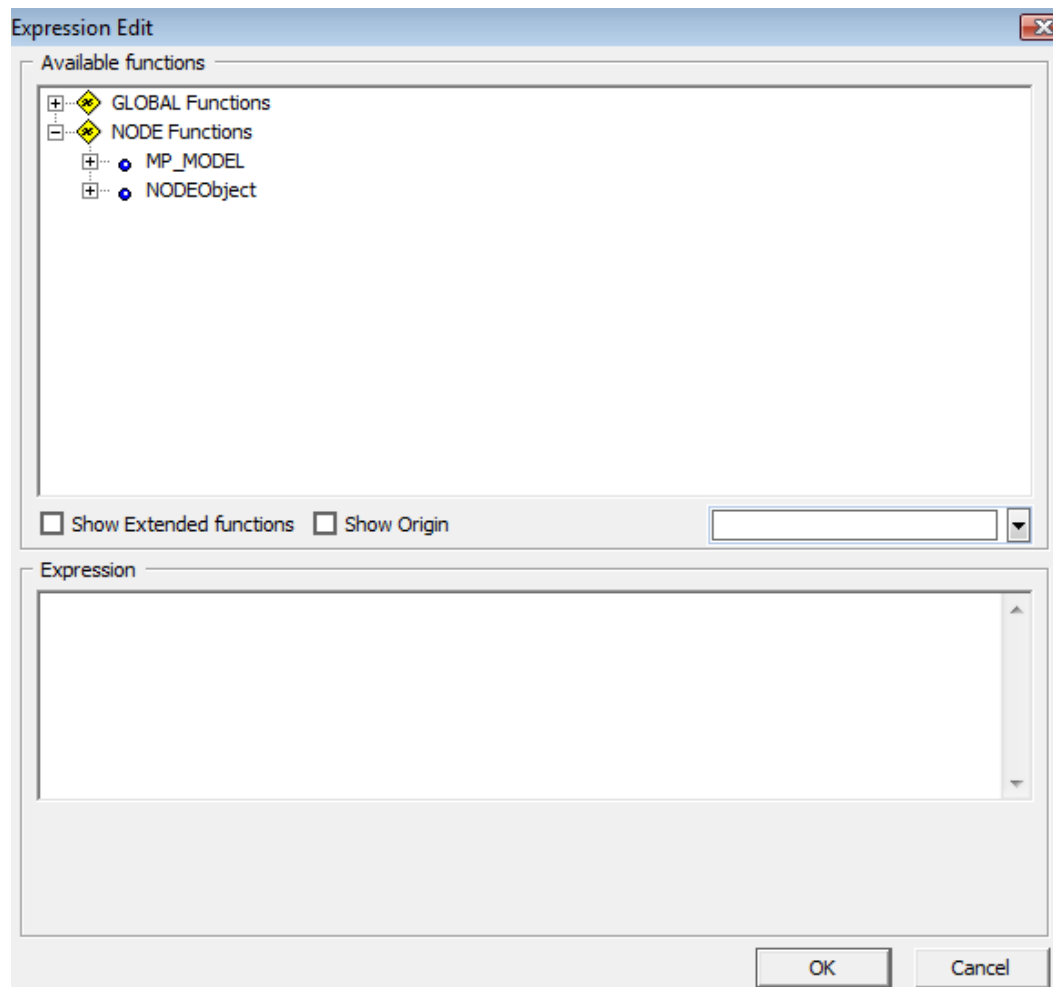


Figure A.3. Expression builder.

Some of the nodes require the user to define an input parameter entry that will be used later during running of the node. These parameters are specified with particular dialog boxes having combo boxes, radio buttons, or text boxes over the window. An expression editing dialog box seen in Figure A.3 appears by clicking on some of the text boxes. User can use mathematical functions, string manipulation functions, time related function, etc. in the expression by selecting under “GLOBAL Functions”. “NODE Functions” is, on the other hand, contains the functions specific to the current node. Except “NODEObject”, the objects or lists under “NODE Functions” are both the fixed and dynamic input link points of the node. While the names of the fixed input link points are listed automatically, the names of the dynamic input link points do not appear until the user create them. All the related attributes and functions of these objects and lists can be reached via this tree structure. It is possible to navigate on the objects or lists by dot operator. In order to formulate objective functions and constraints within GSAMS environment, expression editing dialog boxes are used.

There are also special functions for a list of objects. The most commonly used one is “LISTSUM” function. It is used for manipulation of a list by iterating all objects within the list to perform a task on each object. The list is provided prior to the declaration of function, and separated with a dot operator (“.”). “LISTSUM” function requires defining two arguments which are separated with the comma operator (“,”). First argument indicates any iterator to represent each object within the list. The name of the iterator does not have any significance in execution so that it can be any bunch of characters. Second argument, on the other hand, is an expression to be executed on the runtime for each of the object in the list. Since the defined iterator name represents each object in the list, the dot operator next to the iterator name allows reaching attributes of this object.

After the necessary links are tied and all the required input parameters of the dialog box are entered by the user, the node is expected to be defined completely. An undefined node has a red cross on its right upper corner. In order to control the validity of the syntax of the expression or type mismatches, user can operate error checking mechanisms such as by sending messages.

After an algorithm is defined appropriately, it can be executed by loading the required data. As mentioned previously, this data should be an instance object of the class written within the brackets on the “Base Object” node. The system manager object will be manipulated by the execution of this node, and at the end, the system manager object at the output link point will be available to be used within the algorithm. Then the nodes in the algorithm are executed in a sequence which is organized internally by the system. A node can be run several times in an execution of algorithm by constructing loops. After a node is executed, objects are fired from the link points of that node and reach to related input link points of other nodes via links.

One of the most important flexibilities of the system is to allow using previously modeled algorithms within other algorithms as a simple node. In order to achieve it, user should firstly create an algorithm, and then by selecting this algorithm in the data model under “Models” explorer, he should drag and drop it into the opened algorithm similar to add a regular node. The first algorithm will be seen as a node in the opened algorithm. For example, in the “CuttingStock_Algorithm” seen in Figure A.2, “MasterModel”, “Sub Model1”, and “SubModel2” are the algorithms created previously by the user just like creating a regular algorithm, and then used in this opened algorithm as a node. When a simple algorithm with “Base Object” at the beginning and with “Done” node at the end is used as a node, it will have a fixed input link point, “Object”, which requires an object of the class defined in its “Base Object”. In addition, there will be its corresponding fixed output link point which fires the manipulated “Object” object. However, like in this example, if additional fixed input/output link points are needed, the appropriate input/output nodes under the “Basic Nodes” can be used. For example, the “SubModel1” node within the algorithm has an “Input Object” and “Output Object” of class “MP_Column”, and an “Output Object” of class “MP_MODEL”. As a result, algorithms can be used as a node with several input and output link points within any other algorithm. By that way, algorithms can be decomposed hierarchically into sub-algorithms within GSAMS environment. This is especially efficient in modeling of a big algorithm so that some parts of the algorithm can be grouped into different algorithms and then they can be embedded into the main algorithm as nodes. This will ease to follow up the whole process of the main algorithm.

REFERENCES

- Atamtürk, A. and M. W. Savelsbergh, 2005, "Integer Programming Software Systems", *Annals of Operations Research*, Vol. 140, pp. 67-124.
- Ateji Company, 2000, OptimJ: Java-based Modeling Language for Optimization, [http://www.ateji.com/vpAvt/OptimJ%20 manual%20v1.3.14.pdf](http://www.ateji.com/vpAvt/OptimJ%20manual%20v1.3.14.pdf), February, 2011.
- Balas, E., S. Ceria and G. Cornuejols, 1996, "Mixed 0-1 programming by lift-and-project in a branch-and-cut framework", *Management Science*, Vol. 42, pp. 1229-1246.
- Bazaraa, M. S., J. J. Jarvis and H. D. Sherali, 1990, *Linear Programming and Network Flows*, John Wiley and Sons, New York.
- Benders, J. F., 1962, "Partitioning procedures for solving mixed-variables programming problems", *Numerische Mathematik*, Vol. 4, No. 1, pp. 238-252.
- Bertsimas, D. and J. N. Tsitsiklis, 1997, *Linear Optimization*, Athena Scientific.
- Birgören, B., 1995, AREMOS: An Object Oriented Interactive LP Modeler for Production Planning in a Petroleum Refinery, M.S. Thesis, Boğaziçi University.
- Bisschop J., 2011, A Cutting Stock Problem, *AIMMS Optimization Modeling*, pp. 235-244.
- Brooke, A., D. Kendrick, A. Meeraus and R. Raman, 1998, *GAMS: A User's Guide*, GAMS Development Corporation.
- Chvátal, V., 1983, *Linear Programming*, Macmillan .

- Cordier, C., H. Marchand, H. Laundy and L. A. Wolse, 1999, "bc-opt: A branch-and-cut code for mixed integer programs", *Mathematical Programming*, Vol. 86, pp. 335–353.
- Dantzig, G. B. and M. N. Thapa, 1997, *Linear Programming 2: Theory and Extensions*, Springer .
- Dantzig, G. B. and P. Wolfe, 1960, "Decomposition Principle for Linear Programs", *Operations Research*, Vol. 8, No. 1, pp. 101-111.
- Dash Optimization, 2004, *Xpress-Mosel Language Reference Manual*.
- Debarbieux, D., 2009, Tutorial: Column generation with OptimJ and Lp_solve, <http://forums.ateji.com/viewtopic.php?f=7&t=49>
- DIP, 2010, COIN-OR, <https://projects.coin-or.org/Dip>.
- B. Dominguez-Ballesteros, G. Mitra, C. Lucas and N.-S. Koutsoukis, 2002, "Modelling and solving environments for Mathematical Programming (MP): a status review and new directions", *Journal of The Operational Research Society*, Vol. 53, No. 10, pp.1072-1092.
- Draman, M., İ. K. Altınel, N. Bajgoric, A. T. Ünal and B. Birgören, 2002, "A Clone-based graphical modeler and mathematical model generator for optimal production planning in process industries", *European Journal of Operational Research*, Vol. 137, pp. 483-496.
- Draman, M., İ. K. Altınel, N. Bajgoric, A. T. Ünal and B. Birgören, 2000, "An Object-oriented graphical modeler for optimal production planning in a refinery", in *Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research*, pp. 263-278.

- Draman, M., İ. K. Altınel, N. Bajgoric and B. Birgören, 1995, "Linear optimization and sensitivity analysis by a visual interactive modeler and expert system for production planning in a petroleum refinery", Proceedings of The Tenth International Symposium on Computer and Information Sciences, pp. 481-488.
- FICO XPress Optimization Suite 7.1., 2009, "BCL", FICO XPress Optimization Suite whitepaper, pp. 4-5.
- FICO XPress Optimization Suite 7.1., 2011, FICO, <http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx>, February, 2011.
- Fourer, R., 1983, "Modeling Languages versus Matrix Generators for Linear Programming", ACM Trans Software, Vol.9, pp.143-183.
- Gagliardi, M. and C. Spera, 1997, "BLOOMS: A Prototype Modeling Language with Object Oriented Features", Decision Support Systems, Vol. 19, pp. 1-21.
- Galati, M., 2010, Decomposition Methods for Linear Integer Programming, Lehigh University.
- GAMS Solvers, 2011, GAMS, <http://www.gams.com/solvers/index.htm>, March, 2010.
- Gerişkovan, A. İ., 1999, Analysis Design and Implementation of an Object Oriented Mathematical Modeler, M.S. Thesis, Boğaziçi University.
- Greenberg, H. J., and F. H. Murphy, 1992, "A comparison of mathematical programming modeling systems", Operations Research, Vol. 38, pp. 177-238.
- Hentenryck, P. V., and L. Miche, 2000, OPL Script: Composing and Controlling Models. Lecture Notes in Artificial Intelligence (LNAI 1865), pp. 75-90, Springer Verlag.

- Huh, S. Y. and Q. B. Chung, 1995, "A Model Management Framework for Heterogeneous Algebraic Models: Object-Oriented Database Management Systems Approach", Omega International Journal of Management Science, Vol. 23, No.3, pp. 235-256.
- Hultberg, T. H., 2003, A Presentation of C++, University of Aveiro.
- ICRON Technologies, 2010, <http://www.icrontech.com>, March, 2010.
- ILOG, I., 2008, CPLEX 12.2 User Manual, Gentilly Cedex, France.
- Jünger, M., and S. Thienel, 1998, "Introduction to ABACUS—a branch-and-cut system", Operations Research Letters, No. 23, pp. 83-95.
- Kalvelagen, E., 2002, Column Generation with GAMS, <http://www.amsterdamoptimization.com/pdf/colgen.pdf>, March, 2011.
- Köseoğlu, D., 1999, Object-oriented development of a decision support module for parametric analysis for linear programming, M.S. Thesis, Boğaziçi University.
- Lahdelma, R., 1994, An Object-Oriented Mathematical Modeling System, Ph.D. Dissertation, Systems Analysis Laboratory, Helsinki University of Technology, Otakaari, Finland.
- LINDO Systems, I., 1999, LINDO API 6.1 User Manual, Illinois, Chicago.
- LINGO 10.0., 2009, LINDO Systems Inc., http://www.lindo.com/index.php?view=article&catid=25%3Aproduct-archives&id=53%3Aarchives-lingo-100&option=com_content&Itemid=34
- Linninger, A. A., S.Chowdhry, V. Bahl, H. Krendl and H. Pinger, 2000, "A Systems Approach to Mathematical Modeling of Industrial Processes", Computers and Chemical Engineering, Vol. 24, pp. 591-598.

- Lukasiewicz, M., 2008, Java ILP - Java Interface to ILP Solvers, <http://javailp.sourceforge.net>.
- Martin, A., 1998, Integer programs with block structure, Habilitation Thesis. Technical University of Berlin.
- Maximal Software, 1993, MPL Modeling System User Manual, Maximal Software Inc.
- McKay, K. N., D. B. Kletter and S. C. Graves, 1997, "OMAC: An Environment for Modeling and Analysis", *Annals of Operations Research*, Vol. 72, pp. 241-264.
- Nemhauser, G. L., M. W. Savelsbergh and G. C. Sigismondi, 1994, "MINTO: a Mixed INTegeR Optimizer", *Operations Research Letters*, No. 15, pp. 47-58.
- Pulp Documentation Team, 2010, PULP Documentation Release 1.4.6.
- Ralphs, T. K., and M. Güzelsoy, 2005, "The SYMPHONY callable library for mixed-integer linear programming", *Proceedings of the Ninth INFORMS Computing Society Conference*, pp. 61-76.
- Roelofs M. and J. Bisschop, 1993, AIMMS User Guide, Paragon Decision Technology.
- Sungur, I., 2003, Design and Implementation of Object Oriented Mathematical Programming Environment Based on Object Oriented Framework for Optimization Systems, M.S. Thesis, Boğaziçi University.
- Tebboth, J. R., 2001, A Computational Study of Dantzig-Wolfe Decomposition, Ph.D Thesis, University of Buckingham.
- Ünal, A. T., M. Draman, İ. K. Altınel, N. Bajgoric and A. İ. Gerişkovan, 2002, "Clone-based modeling for optimal production planning in process industries", *International Journal of Production Research*, Vol. 16, pp. 4041-4058.