AREMOS: AN OBJECT ORIENTED VISUAL INTARACTIVE LINEAR PROGRAMMING MODELER FOR PRODUCTION PLANNING IN A PETROLEUM REFINERY

by

Burak Birgören

B.S. in I.E., Boğaziçi University, 1992

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

Boğaziçi University
1994

ACKNOWLEDGMENTS

I would like to express my deepest gratitute to my thesis advisor, Assoc. Prof. İ. Kuban Altınel for his invaluable guidance, support, motivation and patience not only during the thesis but throughout my graduate study. I would like to thank him especially for creating an opportunity for me to conduct a comprehensive research, by which I gained a lot of experience. It was a pleasure to work with him.

I am also deeply thankful to Assistant Prof. Murat Draman for his invaluable guidance and advice in the software design and implementation phases of my thesis.

I would like to thank to Assoc. Prof. Levent Akın for his valuable comments and suggestions as a member of my thesis committee.

I wish to thank great many people. Among them I should mention Halim Karabekir, Celal Doğan and Fikret Yeletayşi of Turkish Petroleum Refineries Corporation (TÜPRAŞ) for their friendly help and valuable suggestions. I wish to express my special thanks to Rasim Mahmutoğulları for his kind help in the interface implementation of AREMOS in the Windows operating system; Dr. Nijaz Bajgoric for his valuable comments on decision support systems.

Finally, I would like to express my deepest gratitude to all of the members of my family. My parents gave me a great moral support and encouragement in my studies. My special thanks go to my sister, Gülüm, who helped me a lot in preparing this document.

ABSTRACT

A prototype graphical modeling system, AREMOS (A Refinery Modeling System), has been developed for modeling the production in a petroleum refinery as a linear program within the scope of this study. AREMOS is an object oriented modeling system which provides a visual interactive environment for refinery modeling. Its aim is to support the management of a refinery in deciding the optimum production policy through a friendly user interface, which guides the user throughout the decision making process, and virtually eliminates the need for mathematical programming expertise.

ÖZET

Bu çalışmada bir petrol rafinerisindeki üretimi bir doğrusal program olarak modellemek amacıyla AREMOS (A Refinery Modeling System) adı verilen bir prototip grafik modelleme sistemi geliştirilmiştir. AREMOS nesne tabanlı bir yazılım sistemidir ve rafineri modellemesi için görsel etkileşimli bir ortam sağlamaktadır. Sistemin amacı en iyi üretim politikasının saptanmasında görsel etkileşimli bir kullanıcı arayüzü yoluyla rafineri yönetimine yardımcı olmaktır. Sistemin kullanıcı arayüzü, karar verme sürecinde kullanıcıya kılavuzluk etmekte ve kullanıcının matematiksel programlama deneyimine gereksinimini en aza indirmektedir.

TABLE OF CONTENTS

Page

LIST OF FIGURES

1. INTRODUCTION

TÜPRAŞ, Turkish Petroleum Refinery Corporation, is composed of four refineries; namely İzmit, İzmir, Orta Anadolu and Batman refineries. TÜPRAŞ İzmit Refinery, the biggest among the four, processes more than 10 million tons of crude oil each year which amounts to almost half of the Turkish yearly crude oil consumption. It is obvious from this very fact that an effective and efficient production planning for this refinery is crucial with regard to its role in Turkish petroleum industry.

Computer-based mathematical programming techniques have been used and proved to be very successful in the petroleum industry applications for more than 30 years. It is a worldwide practice in petroleum refineries to use such techniques for production and process planning, and strategic planning. TÜPRAŞ is not an exception; TÜPRAŞ management has been using these methods for a long period of time. At present, a software package called the Process Industry Modeling System (PIMS), developed by Bechtel Inc., is used in TÜPRAŞ İzmit refinery for operations and production planning. Moreover a decision support system, called the Supply Options Optimization System (SOOMS) was developed a few years ago for the purpose of evaluating capital investment and strategic planning decisions including all TÜPRAŞ refineries.

On the other hand, it is hard to say that the management is fully utilizing these management tools. The main reasons for this seem to be the lack of expertise on mathematical programming and lack of effective communication with the existing mathematical programming system. Consequently, it has turned out to be difficult to handle the problems resulted from the improper feed of data into the mathematical modeling system and the interpretation of system solutions. Besides, the refinery management has faced important problems in updating the existing system according to the changes in the refinery. Therefore, the maintenance of the mathematical programming system has appeared to be another important concern.

As a solution to these problems, AREMOS, a prototype optimization based visual interactive modeling (VIM) system, has been developed in this study. VIM methodology, which is currently gaining wide acceptance in Management Science/Operations Research (MS/OR) applications, aims to provide friendly, interactive and graphic support to decision makers. In other words, it puts much more emphasis on the user side than the former decision support tools.

The system has been designed using an object oriented approach. The main objective of this selection is that object oriented programming is considered to be an obvious vehicle for the development of visual interactive system packages. In addition, it is a solution for the

software maintenance problem, since object oriented nature of a system considerably facilitates the maintenance process.

In order to implement the object oriented design, the C++ language is selected as the programming language and the Microsoft Windows operating system is selected as the software environment. The reason for this selection is not only their popularity but also the fact that they provide a very low cost platform for software development. Further, additional data structure libraries and Windows programming class libraries are used.

Although the starting point of this study is based on the TÜPRAŞ İzmit Refinery case, all the possible processes and products in a typical petroleum refinery are included in the design, thus the developed system can be applied to any refinery with minor changes.

The interface of AREMOS provides a visual representation of the refinery in the form of a network, on which refinery process units are represented as nodes and product flows as arcs. It enables the user to build the refinery visually on the screen, feeding all the necessary data through dialog boxes. The system generates the corresponding mathematical programming model in the background, sends it to an optimizer and gets the solutions, and presents the necessary optimal production results and their interpretations to the user. It also helps by giving comments on errors and inconsistencies in the refinery building process and on the results. The system virtually eliminates the need for mathematical programming modeling expertise, and further, the need for any mathematical programming language.

AREMOS is a prototype system, which needs to be tested by the refinery management to detect its weaknesses and strengths. According to the comments of the refinery management, it can be improved by adding more visual utilities and more powerful mathematical programming techniques. On the other hand, it satisfies the basic requirements of ease of use, and ease of maintenance to a high degree.

2. LITERATURE SURVEY

The literature survey for this study has been conducted on five different topics :

(a) Mathematical programming applications in the petroleum industry;
(b) Mathematical programming languages and systems;
(c) Visualization in optimization;
(d) Visual interactive modeling methodology;
(e) Object Oriented Programming.

First, sufficient knowledge in previous mathematical programming applications in the petroleum industry is very important for this study since it will constitute the base for developing a visual interactive modeling system. The related literature survey is mostly based on refinery applications and covers the period of last 40 years. Second, mathematical programming languages and systems are surveyed in order to assess the applicability of different mathematical programming paradigms in refinery modeling. The emphasis in this survey is given to the new mathematical modeling languages and systems which make use of newly developed modeling techniques. Since a visual system is the aim of the study, related research in the field of operations research, and specifically in the mathematical programming area is examined. Next, the underlying concepts and the general philosophy of visual interactive modeling are reviewed. Finally, a brief survey on object oriented programming, which concentrates on the topics such as general principles of object oriented programming, the design of an object oriented system and its implementation for visual interactive user interfaces, is performed.

2.1. Mathematical Programming Applications in the Petroleum Industry

The extensive use of linear programming in the petroleum industry is well known and often cited as a major triumph of operations research [1]. The use of mathematical programming in the petroleum industry spans a period of more than 40 years. Many mathematical programming concepts and techniques together with linear programming have

been applied in almost all facets of oil business from strategic planning through process control within this long period.

The usage prior to the invention of the electronic digital computers was limited to small problems that could be solved by hand. As the computer technology advanced, first, linear programming techniques, and then more advanced techniques, such as nonlinear programming and dynamic programming, have been applied in the petroleum industry. Especially in the 1980s, mathematical programming applications in this field became a worldwide practice which yielded very successful results. Bodington and Baker give a detailed history of these applications [1].

The 1950s were a decade of experimentation. By 1955, some oil companies installed stored program electronic digital computers and experimented with linear programming. For example Symonds gives an overview of these studies based on the applications in ESSO Oil Company in a paper published in 1956 [2]. Besides, the studies conducted by Charnes, Cooper and Mellon [3], Manne [4], and Kawaratani, Ullman and Dantzig [5] can be given as the examples of the early research in this field. By the end of the 1960s other linear programming, successive linear programming and nonlinear programming models were developed to model different phases of the oil business, which included production planning, process control, product blend optimization, product distribution and marketing.

In the 1970s model management became the key issue of research since the main difficulty of the companies, especially involved in large scale refinery modeling, lay in model management, for example database management, model generation and solution reporting. Software and consulting firms began to provide model management tools for the petroleum industry. In this period models with broader scope were developed and all the techniques and applications were enhanced in speed, capacity and efficiency.

After the 1980s, computers became powerful enough to solve large refinery problems in reasonably small times, and also database technology finally came of age. Consequently, computer applications of mathematical programming models of refineries became both efficient and effective. As a result, the petroleum industry experienced extensive applications of mathematical programming by operations researchers. To cite a few of them; Baker and Lasdon made an application of successive linear programming at Exxon Oil Co. in 1985 [6]. Two years later, Klingman et al. developed an optimization-based intelligent decision support system implemented at Citgo Petroleum Corporation to address the complex short-term planning and operational issues associated with the supply, distribution, and marketing of refined petroleum products [7,8]. Another research, conducted by Dewitt et al. in Texaco Oil Co., focused on gasoline blending which is a key refinery operation [9].

Almost all of these applications were based on mainframes and related software. However there has been a general trend toward microcomputers from mainframes since the

end of the 1980s. Uhlmann discussed the applicability of linear programming modeling for refineries on microcomputers in 1988 [10]. Refineries are big industrial complexes and require big algebraic models. Uhlmann gave the size of a linear programming based refinery model for an average size refinery as having a generated matrix of 717 rows, 2168 columns and 333 bound elements with a density of 0.77 per cent. It is reported that the model was solved in 414 minutes on an IBM PC/AT machine, which is quite a long time. Consequently big sizes and long solution times had been the major problems for not using microcomputers. Nevertheless, in the recent years microcomputer technology has experienced a great improvement, and today, microcomputers, which are at least 20 times as fast as AT machines, are available in the market with reasonably low prices.

There have also been Mathematical programming applications in the Turkish petroleum industry. In an early application, Koşal developed a methodology for production planning in a Turkish refinery in 1981 [11]. Another research was conducted for the development of production planning in İzmit TÜPRAŞ Refinery by Kavrakoğlu et al. in 1986; they developed a single period linear programming model to optimize the production [12]. Four years later, a multi-refinery, multi-period modeling system was developed for TÜPRAŞ including all its refineries [13]. Besides refinery modeling, Gürkan and Kartal applied mathematical programming to model the Turkish petrochemical industry in a recent study [14]. Over and above these scientific studies, it should be noted that usage of mathematical programming packages has been a routine practice in Turkish petroleum refineries for a long period of time.

2.2. Mathematical Programming Languages and Systems

There has been a recent upsurge of new modeling languages and systems for mathematical programming. This stems from three basic reasons. First, early systems were written for specific computing environments and were not immediately adapted to modern programming environments that emerged in 1980s. Second, a new generation of modelers and managers became dissatisfied with the perceived complexity of the early systems; in particular, these had and still have the requirements of a programmer's skill level. Third, demands for computer assisted modeling and analysis increased not only with the need, but also with new technologies that render such demands achievable, notably database concepts,

artificial intelligence and graphics. Greenberg [15] gives a detailed annotated bibliography of the recent studies in this area.

An important issue in the development of mathematical programming systems is model management which is a main concern of not only industrial engineering but also many computer related disciplines. Krishnan [16] gives an overview of model management research on mathematical programming systems. He presents an analysis of the model management literature, focussing selectively on the tasks that are currently a major focus of research. This includes topics such as model selection, model composition, model integration, model formulation, model implementation and model interpretation.

An important attempt to bring formalism for modeling is Geoffrion's Structured Modeling [17,18,19,20]. Structured Modeling represents a major effort towards a sound basis for modeling theory and practise. The objective of Structured Modeling is to develop a comprehensive framework to represent all the essential elements of a variety of management science models. It provides a very consistent and complete internal representation scheme. In these works, Geoffrion also explains characteristics of a modern modeling system.

Murphy, Stohr and Asthana [21] review different methods for representing linear programming models during the formulation phase. They make a comparative analysis of these systems from the point of view of the interface presented to the user. There are two major categories of linear programming representation schemes: Those whose underlying philosophy is primarily symbolic, and those whose underlying philosophy is primarily graphic. This main classification can be applied to mathematical programming models, in general. Under symbolic representation schemes lie matrix generators, algebraic languages and database oriented languages. Matrix generators basically provide language statements that produce MPS statements directly. The objects to be manipulated are sets, and data tables and the end result is a list of data triples in MPS format. MPS is the common linear programming interchange format among different linear programming languages. Examples of matrix generators include OMNI, DATAFORM and GAMMA [21]. Algebraic languages represent the structure of a model using a notation that closely resembles conventional mathematical notation. GAMS [22] and AMPL [23,24] can be given as the examples of these languages that are widely accepted in the software market. LINDO [25] can also be regarded as an algebraic language although it allows a very restricted form of algebraic input in extended coefficient form, allowing no summation or indices. Database oriented languages, on the other hand, view a linear program as a relational database tables and extend conventional query languages to express algebraic constructs such as summation over sets. SQMPL [26] is an example for such languages.

Graphic representation schemes comprise structured modeling, block-schematic languages, network representations and iconic languages. Structured modeling [17] takes an algebraic view, but classified as graphic since it emphasizes several different graphs that

relate algebraic and data elements in the problem description. Geoffrion has developed the SML language for structured modeling [27,28]. In block-schematic languages, the graphic elements that the model builder manipulates conceptually are blocks within the linear programming matrix together with icons representing their interconnections. MathPro is a block-schematic language available in the market [29]. Among network representations, an activity-constraint graph has nodes for activities and constraints and links that connect these nodes to show which activity participates in which constraint and vice versa. As another network representation, netform graphs represent networks of activities with one input and one output in which the nodes represents constraints and the arcs represent flow of resources. Another promising scheme is iconic languages which attempt to use graphical objects that are analogous to real life objects. LPFORM is a prototype iconic system for linear programming model development. The interface for the LPFORM system uses icons to represent real world objects such as inventories, machines and transportation networks [30,31]. It should be noted that there is still no graphic language which has gained acceptance in the market like GAMS or AMPL.

Finally, Murphy and Greenberg [29] make a comparison of mathematical programming systems, classifying them as traditional, algebraic and block-schematic. They choose OMNI, GAMS and MathPro as three representative languages for these categories respectively, and formulate four linear programs in each language. According to their experience they make a comparison among these languages using different modeling aspects. They comment that there is no dominant approach. Each of the systems supports a valid modeling philosophy and is well designed. Each system also has distinctive capabilities and features that attract users.

2.3. Visualization in Optimization

Research in optimization has traditionally concentrated on analyzing complicated problems and developing faster and faster algorithms to solve them. However solving problems using optimization techniques involves more than just developing clever algorithms. Building, debugging, validating, and understanding models, algorithms, data, and solutions require appropriate representations, in other words, visualizations. Operations researchers have not been focused on the representations that people actually used to help solve problems until recently. However, today's advanced computer technology provides

very easy-to-use computer programs with very effective user interfaces, which enables operations researchers to develop appropriate visualizations to help model builders, algorithm designers as well as nontechnical specialists. Consequently brand-new user interface techniques have been emerged and applied in the field of operations research in the recent years. Jones surveys and makes a long critique of the ongoing researches on visualization in this field in two feature articles [32,33].

Jones examines the studies using primarily two points of views; modeling life cycle in a typical optimization problem solving process and available visualization formats. The modeling life cycle is composed of the following stages; conceptualization, formulation, data collection, solution (algorithm execution), solution analysis and results presentation. For all of these stages a number of visualization techniques have been developed. To cite a few of them, high level languages, formal and informal diagrams and visual interactive modeling as a new methodology are used to describe the conceptual model at the first stage. Object oriented programming and spreadsheets as well as the mathematical programming languages outlined in the previous section are used in the formulation stage to give an appropriate representation of the model for formulation. Relational databases and spreadsheets are the most commonly used tools to provide linkages between commercial mathematical programming systems and the data for the problem instance. The representations developed for the solution stage aim to promote the exploration of internal algorithm execution. These include human intervention during the solution process, visualizations to help understand the execution of the algorithm, as well as visualizations that provide new and novel insights. In this context, related techniques are dubbed "interactive optimization" and "algorithm animation". Solution analysis and results presentation include a long process of debugging and validation. Much research have been conducted to provide better analysis and presentation of solutions, and many visual techniques have been used for this purpose such as animation, graphic plots, hypertext and hypermedia as well as natural language.

As another point of view, available visualization formats can be examined separately. Software technology provides a number of powerful techniques to be used in many scientific disciplines and promise much advanced ones in the near future. These techniques include hypertext, diverse graphic techniques such as pie charts and bar charts, animation, sound and touch utilities and virtual reality. In addition to these, text can also be regarded as a visual technique. For example, Geoffrion [27,28] in SML language showed how different text styles could be used to highlight different aspects of information.

Jones also discusses how different representation formats can be integrated with different tasks involved in the modeling life cycle [33]. He argues that different users require different representations according to experimental evidence, and exactly which representations will be appropriate for particular types of users and tasks remain poorly understood. Many proposals for different representations for mathematical programming

exist. Yet, given the variety of representations, it does not seem likely that one representation will come to dominate. Rather different representations will coexist. To support multiple, simultaneous representations, window-based user interfaces were developed and are now common. For example Apple Macintosh, Microsoft Windows and X Windows are equipped with such capabilities. Any modern interface for mathematical programming will involve a multiple window user interface. On the other hand, commercial optimization tools that exploit this technology are only now emerging, even though it has been widely available since the mid 1980s.

In short, visualization helps solve problems. Numerous examples exist to support this claim. Exactly what problems and what visualizations are most helpful for particular tasks, and particular people, including researchers, practitioners and consumers of optimization, remains difficult to assess and, hence, an area for ongoing research. Nevertheless, emerging technologies hold promise for expanding the range of visualization techniques employed in optimization. Equipped with these interfaces, operations research tools will be likely to provide much efficient and effective means of solving problems.

2.4. Visual Interactive Modeling

Visual Interactive Modeling (VIM) has emerged as an MS/OR methodology. It is a general methodology for creating appropriate model representations. As an important characteristic, VIM approach gives prominence to conceptual model representations [33]. In this respect, it can be thought of in terms of a larger movement called User-Centered Design which spans the fields of human factors, ergonomics, and industrial design. Advocates of user-centered design believe that designers should begin, continue, and end the design process focused on users' needs. Adopting these general ideas, VIM starts with the user and user's problem, not with the model. Therefore VIM does not concentrate on a particular technique and is not attached to a particular set of modeling constructs, however general. Rather, it first attempts to represent the actual problem, usually visually. At that point, the model and the solution technique can be developed. Unfortunately, the VIM methodology makes it more difficult to assess the quality of the solution produced, since taking the user centered approach espoused by VIM, forces one to measure the quality of the solution by less objective measures such as user satisfaction.

VIM has visualization and interactivity as its two important characteristics. In the previous section, what visualization brings into the problem solving process is discussed. Interactivity can be considered as an inevitable extension of visualization. Interactivity together with visualization primarily integrates the user into the problem solving process, letting him to control the process from the beginning to the end, which makes the modeling system more user-friendly and effective. Today interactivity is accomplished through mouse-driven or pen-driven operating systems such as Microsoft Windows. Developments in the computer technology suggests that the audio-visual interactive modeling systems are not far away, in which natural speech will be a part of interaction between the user and the computer.

Bell gives a history of VIM with the present situation and the prospects [34]. In his paper, he gives a detailed list of VIM applications. Bright and Johnston [35] approach the subject from another perspective, discussing the intrinsic nature of problems which makes them susceptible to the VIM approach. They argue that VIM allows the user to redefine the problem, the objectives and the solution space quickly. It loosen the constraints and promotes reconsideration of objectives. More importantly, it allows the problem owner to assess the effects of other objectives. Then they discuss the features that characterize the nature of VIM software, that are speed, adaptivity, width of application and ease of use. An interesting issue in this field is the integration of intelligence in VIM. In a recent paper, Hurrison describes methods by which expertise may be added to visual interactive models [36].

2.5. Object Oriented Programming

The software systems built today are different from what they were ten or twenty years ago; they are larger, more complex, and more volatile. Object oriented programming aims to manage this increasing complexity providing a number of interrelated principles. These are data abstraction, data encapsulation, inheritance and polymorphism [37]. Today, object oriented approach is applied to software development, starting from the analysis of the real system, through the design of the software to the end of the implementation process [38,39].

Object oriented programs model a system as composed of objects and the interrelations among them. The programs perform computations by passing messages

between active objects, which are computer analogs of entities in the real world. Objects in a program belong to classes, in other words an object represents the instance of a class. In an object oriented school database program, for example, the file of a specific student can be viewed as an object belonging to the student class. The necessary data is packaged in a class with the methods that apply to them. The methods of a class describes the behavior of the class for treating the received messages. This concept is called data abstraction. Further, the interface to a class, that is the methods of the class, is defined in such a way as to reveal as little as possible about its inner details, which is called data encapsulation or information hiding. Another important principle is inheritance, which is the ability to define a new class that is just like an old one except for a few minor differences. This encourages the programmer to group similar objects according to common attributes and define specializations with added local detail, which also help building complex models hierarchically. Last of all, polymorphism is the ability of objects from different classes to respond differently to the same message. Polymorphism is partly responsible for a well known characteristic of object oriented systems; a style of programming sometimes referred to as differential programming or programming by modifications. It makes it very easy to plug new objects into the system if they respond to the same messages as existing ones.

As an important benefit of these complementary principles, object oriented programming improves the productivity in software development process. It decreases the time required for testing and removing the programming errors. On the other hand, much of the functionality of a software system is added after the initial deployment, which implies the importance of maintenance. Object oriented programs also enable programmers to make modifications and improvements more quickly. As another benefit, object oriented approach lets programmers reuse classes in new projects without reinventing them. Briefly, an object oriented approach to software system development is likely to lead to more stable and robust systems, with promoted maintenance and productivity in the development process.

In addition to these benefits, today's on line, interactive systems devote a great attention to the user interface, and object oriented approach to such systems -from analysis through design and into coding- is a more natural way of dealing with such user oriented systems [39,40]. It is also argued that object oriented languages are suitable for VIM packages [34]. There are already software development tools and platforms for the development of object oriented user interfaces. ObjectWindows for C++ [41], for example, is such a platform supported by extensive class libraries for interface design under Microsoft Windows operating system.

So far object oriented VIM has found its application mostly in simulation. Object oriented methodology has been applied just in a few mathematical programming systems. ASCEND [42], for example, is a model building environment for complex models comprising large sets of simultaneous nonlinear algebraic equations, which uses and extends

object oriented concepts. In this respect, integration of the concepts from VIM and object oriented programming is a fairly new research area.

3. PRELIMINARY WORK

As a preliminary work for this research, a linear programming model for the TÜPRAŞ İzmit Refinery is developed in cooperation with the refinery management [43]. This study is presented in the XVth Turkish National Operations Research and Industrial Engineering Congress and published in the congress proceedings [44]. In this work, the linear programming model is generated and solved using LINDO optimization software package. The model maximizes the daily net profit, and computes an optimal production plan: It basically computes the amounts of crude oils to be fed into the system and final products to be produced, refinery unit utilizations, and utility consumptions such as electricity and different catalysts. A sensitivity analysis is also performed for different scenarios.

In this study, the model is composed of 218 variables and 171 constraints. The variables of the model represent the quantities of material flowing at each stage of the refinery process: Crude oils, semi-finished products and final products. The quantities are expressed in volume for all the variables. The objective function consists of sales revenues, crude oil costs and utility costs. The constraints are the mathematical expressions of material relationships within the refinery. These are mainly material balance constraints, unit capacity constraints, product specification constraints and demand pattern constraints.

On the whole, the developed model is detailed enough to describe the refinery's production, furthermore, the validation of the model provided strong insights that the results obtained through the model are consistent with the real production figures of the refinery.

The same mathematical model is also developed using GAMS. The main purpose of this application is to check whether an algebraic mathematical programming language like GAMS or AMPL brings any additional ease into modeling. Using GAMS, related variables can be grouped through the use of indices, hence general categories for constraints can be formed. This generalized formulation process is the main advantage of GAMS [22]. However when applied to the linear programming model of the refinery, the number of constraints is not reduced to a great extent in the new model, since many constraint categories and related data tables appear to be formulated. The only considerable reduction in the number of constraints is obtained when formulating the constraints for distillation process. Furthermore, the formulation process turns out to be time consuming, and in this respect there is no significant difference between LINDO and GAMS. Nevertheless, using GAMS brings about a broader view for the refinery model; variables and constraint types generated in this modeling study can be used while forming the basis for the VIM system.

Finally, the current study on the development of AREMOS prototype refinery modeling system is presented in the XVIth Turkish National Operations Research and Industrial Engineering Congress and accepted for publication in the congress proceedings [45].

4. THE SCOPE OF THE STUDY

As briefly stated in the introduction section, this study intends to build a prototype object oriented visual interactive modeling (VIM) system to determine the optimum production plan in a petroleum refinery by means of linear programming.

Better representations for solving problems are becoming the main concern of MS/OR scientists, since end users, in other words decision makers, want to have effective means of using MS/OR techniques. An obvious solution to this problem is to provide decision support tools in the form of advanced computer programs, because, as Jones argues [33], easy-to-obtain, easy-to-use computer programs are the most effective communication link between high-powered research teams and the average professionals hoping to apply the results of research to real life situations.

This type of managerial need is exactly the case in TÜPRAŞ İzmit Refinery, and constitutes the main motivation for this work. The refinery management doesn't have sufficient mathematical programming expertise, thus generally unable to interpret the solutions of the existing mathematical programming based modeling system (PIMS) correctly. Moreover, they have difficulty in communicating with the system, since it restricts the interventions of the user, require data in a strict format, doesn't allow new additions according to changes in the refinery processes. Lack of user friendliness is also a big problem, since the management argues that some modules of the system are not utilized, because these modules are very hard to use. Good maintenance seems to solve some of these problems, but unfortunately, it has not been available so far. Also it is questionable whether the design of the existing system is good enough for effective software maintenance.

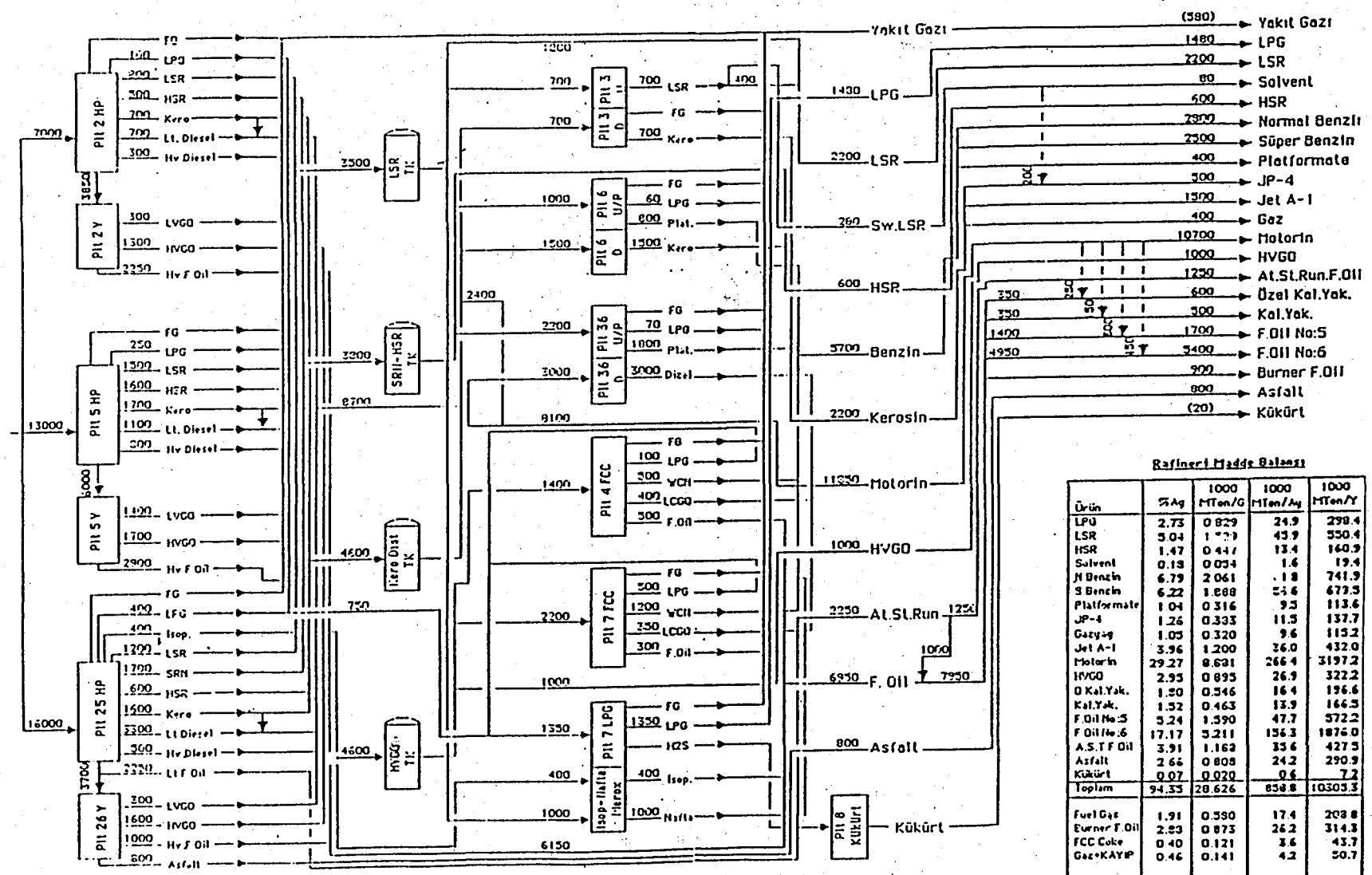
On the other hand, mathematical programming techniques for refinery processes are extensively used and well understood, hence there is no current need for better formulations. Rather, development of an advanced computer system employing such techniques is needed to solve the problems of the refinery management. Considering the nature of these problems it is clear that the emphasis should be given to the user side of such a modeling system. This means a software system with a good user interface, and a robust design which provides easy means to manage the interface, and to update and extend the system without increasing its complexity.

VIM methodology, outlined in the literature survey section, seems to be a good solution for the problems of refinery management. Thus, the question is whether this methodology is appropriate for modeling a refinery as a mathematical program. Based on the preliminary study on TÜPRAŞ İzmit Refinery, our answer at this point is that VIM is fit for this purpose. First, it is clear that the graphical representation of a refinery, namely refinery

process flow chart, is analogous to a network, and our experience is that the management of TÜPRAŞ İzmit Refinery also prefers to use such representation schemes in production and process planning; this is very natural because refinery engineers work on process flow charts, which lets them gain basic maturity for understanding network models easily. Figure 4.1 depicts a process flow chart covering the whole process in TÜPRAŞ İzmit Refinery, which is prepared by the refinery management. In the network analogy of a refinery, units correspond to nodes, and pipes through which products flow correspond to arcs. Second, a typical refinery can be modeled as a mathematical program simply by representing its units (ie. distillation units, tanks and blending units) and flows between these units. Flows are conserved by either mass or volume, and most of the fine petrochemical operations can be approximated by constructing linear relationships between inputs and outputs. In such a refinery representation, a hidden sub-model stands for the internal chemical processes within each node. No other mathematical relations, which are independent of this network representation, are needed in the modeling. Also, success of visual interactive simulation packages, although the underlying optimization paradigm is different, is a source of motivation, since many of them use such network based visual representations for modeling process [34].

The design of the underlying interface management part of the system has almost equal importance as the design of the interface, as it has been the case for all similar software projects. The underlying design should be consistent with the interface, should enable the programmer to easily manage the interface and underlying database, should be flexible for modifications, deletions and additions. These characteristics become more important as the complexity of a system increases, which is the case for the intended system. Object oriented design approach is claimed to provide these characteristics as discussed it the literature survey section, and has already been proved to be successful in visual interactive simulation packages [34].

Bringing these ideas together led to the development of AREMOS modeling system. It is a visual interactive refinery modeling system and designed using the concepts of object oriented programming methodology. The software code of the system is written in C++ which is the most commonly used object oriented programming language in the software market. In addition, Microsoft Windows is selected as the target operating system considering its wide usage in the world, its extensive visual utilities and low price. Consequently, Borland C++ compiler version 3.1, which has necessary utilities for programming under Microsoft.Windows, is chosen for software development, and as a benefit of object oriented programming, extra class libraries are used to facilitate the programming process.



Not : Akımlar m³/gün cinsindendir. Parantez içinde gösterilen akımların birimi MTon/gün'dür.

HAZIRLAYAN : TEKNİK SERVİSLER MÜDÜRLÜĞÜ

Rafineri Madde Balansı

Ürün	%Ag	1000 MTon/G	1000 MTon/Ay	1000 MTon/Y
LPG	2.73	0.829	24.9	298.4
LSR	5.04	1.529	45.9	550.4
HSR	1.47	0.447	13.4	160.9
Solvent	0.13	0.054	1.6	19.4
N Benzin	6.79	2.061	61.8	741.9
S Benzin	6.22	1.888	56.6	679.5
Platformate	1.04	0.316	9.5	113.6
JP-4	1.26	0.383	11.5	137.7
Gazyağ	1.05	0.320	9.6	115.2
Jet A-1	3.96	1.200	36.0	432.0
Motorin	29.27	8.881	266.4	3197.2
HVGO	2.95	0.895	26.9	322.2
Ö.Kal.Yak.	1.80	0.546	16.4	196.6
Kal.Yak.	1.52	0.463	13.9	166.5
F.Oil No:5	5.24	1.590	47.7	572.2
F.Oil No:6	17.17	5.211	156.3	1876.0
A.S.T.F.Oil	3.91	1.163	35.6	427.5
Asfalt	2.66	0.808	24.2	290.9
Kükürt	0.07	0.020	0.6	7.2
Toplam	94.35	28.626	858.8	10303.3
Fuel Gaz	1.91	0.580	17.4	208.8
Burner F.Oil	2.89	0.873	26.2	314.3
FCC Coke	0.40	0.121	3.6	43.7
Gaz+KAYIP	0.46	0.141	4.2	50.7
Hampetrol	100.0	30.341	910.2	10922.7

FIGURE 4.1 The Process Flow Chart for TÜPRAŞ İzmit Refinery

The extra class libraries used in the object oriented design of the AREMOS system are Borland Container class library [46] and Borland ObjectWindows class library [41]. Microsoft Windows is not an object oriented operating system, nevertheless ObjectWindows provides a consistent object oriented platform forprogram development under Windows. Appendix A gives detailed information on these libraries and the compiler.

AREMOS modeling system is not intended to be directly used in TÜPRAŞ İzmit Refinery, but is a prototype, aiming to bring the ideas together from VIM and from mathematical programming modeling for refineries. The system, proposed as a solution to the problems of the management of the refinery, is not claimed to provide the best visual interactive representation scheme, actually there are no objective measures to assess which representation is better [33]. It is rather aimed to establish a sound basis for the development of a real refinery mathematical programming system. In this sense, the object oriented nature of AREMOS has the desired characteristics; it is easily extendible, new modules can be added and old ones can be modified or canceled, it easily manipulates the interface and data, constructing one-to-one relations between interface objects and the corresponding data, and new visual utilities can be easily added provided that they are supported by Microsoft Windows. As an indication of its constituting a sound basis for refinery modeling, it is worthwhile mentioning at this point that the addition of an expert system module and a simulation module as future projects are already under consideration, and this prototype system is being considered to provide a healthy platform for such improvements. However these are not within the scope of this study, and will be cited only as future work.

Finally, AREMOS is designed in such a way that it can be applied to any refinery by quick modifications, hence it is not specific to TÜPRAŞ İzmit Refinery, but rather is a generic prototype object oriented visual interactive linear programming modeler for production planning in a petroleum refinery.

5. THE REFINERY PROCESS AND MODELING

This section gives a brief summary of the refinery process and its mathematical model. For more detailed information, the reader may refer to a recent research paper [43], and to Uhlmann's paper on linear programming on a microcomputer [10]. Besides, Bodington and Baker give an extensive bibliography on mathematical programming in the petroleum industry [1].

The primary inputs of a refinery are different crude oils. Crude oils are supplied by purchase on the market, and hence have a cost. In addition, their procurement is subject to market constraints. Crudes are first processed in distillation units, where components are produced. Two different types of distillation units are used in TÜPRAŞ İzmit Refinery, these are HP distillation units and Vacuum distillation units. The residual remaining after crude distillation in an HP unit is fed into a Vacuum unit which refines the residual into new components,

The components produced by distillation units can be used as feedstock for other processes, or can be blended into end products. Heavy components tend to be reprocessed whereas light ones are more directly blended into end products. The processes for heavy products primarily include chemical reactions with catalysts under high temperatures or pressures. For these processes, yield tables give the nature of component input, nature of component output, and the relationship between inputs and outputs. These processes are performed in FCC (catalytic cracking unit), HCC (Hydro-cracking unit), CCR (Unleaded Gasoline unit), Platformer, Unifier, Desulphurizer, Hysomer and LPG (liquefied petroleum gas processing) units in TÜPRAŞ İzmit Refinery. Apart from the own production of the refinery, components can be imported for processing in these units or for blending.

The final step in oil production is the blending of components into end products. Most of the end products should conform to some specification limits, such as viscosity, sulphur content, density and octane number limitations. The end products within these limitations are obtained by blending different products with different specification levels.

To model the refinery as a linear program, variables, constraints, and an objective function are defined. Variables represent the quantities of material flowing at each stage of the process. Quantities are expressed in volumes, densities are incorporated wherever weights are used. A variable corresponds either to a real product flow or a part of real flow which is needed in modeling blending processes (ie. the amount of product i used in blend j).

Constraints are the mathematical expression of material relationships within the refinery. They express the process description, the quality description and other restrictions.

Process description equations are mostly material balance equations, yield equations or unit capacity equations. These equations describe the whole product flow scheme in a refinery process network. Quality description equations are written for blending operations. Certain specifications are set for products such as minimum and maximum octane levels. Since each blending component has a different level in each type of quality, the blend has to match the specifications for the product. Qualities are blended by either weight or volume: For example, sulphur content specification can be written as a linear equation using weights of the blended products. Restrictions are the limitations on the amount of crude input, import products and final products. There are more complicated restrictions which are not implemented in this study. For instance, in some cases the proportion of each crude, or a combination of crudes in the total feedstock, is limited according to certain regulations. Also there are limitations put on the rated combination of components blended into one single product as a percentage of the total blended weight [10].

The objective function driving the linear programming model is the maximization of the daily net profit for the refinery. The results are projected over the planning horizons, for instance 3 or 6 month periods. Only variable incomes and profits are included in the derivation of the daily net profit. Incomes are essentially due to the sales of products. Costs are the prices paid for raw materials (crude oils) and components, and the costs of running the refinery process units.

AREMOS lets the user model the refinery production visually on the screen without any need of refinery mathematical programming knowledge. The user should only have basic refinery engineering knowledge to construct the refinery model and enter the associated data. AREMOS maintains a hidden object oriented model of the visual refinery model, and generates the mathematical constraints and objective function through the objects of the hidden model. Subsequent sections explain this process in detail..

6. A REVIEW OF THE MODELING SYSTEM

The refinery modeling system AREMOS can be viewed as if it were composed of 3 parts: The refinery modeling interface, the underlying object oriented management system and the optimization module. These are illustrated as three layers of the modeling system in Figure 6.1.

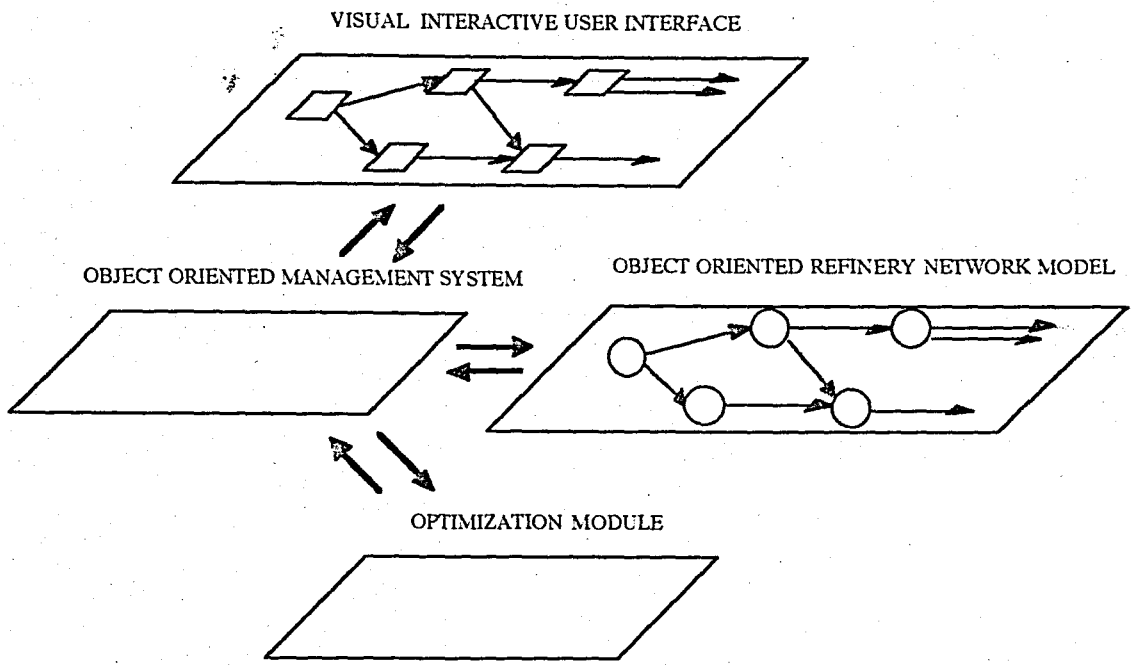


FIGURE 6.1 A Layered Representation of the General Structure of AREMOS

The interface part basically provides the user with a visual refinery model development platform. A visual refinery model is composed of refinery process units and products flowing among the units as illustrated in Figure 6.1. The user develops such a model by means of clicking the mouse on the screen objects which represent units and flows, and selecting menu options provided in the menu bar of the interface. How the interface is used will be discussed in detail in the following section.

The object oriented management system is at the core of the AREMOS system. It is responsible for the management of all the system's functionality. It has interface

management, database management and optimization management duties. As depicted in Figure 6.1, it maintains an internal object oriented representation of the visual refinery production model. This representation consists of refinery unit objects, product flow objects and the refinery object which maintains units and product flows. These objects, in turn, have interface, database and optimization functions. The object oriented management system performs its duties by means of the functions of these objects. The object oriented management system retrieves the user messages from the interface, such as menu selections and mouse clickings, and gives proper responses. It communicates with the objects of the refinery network model to perform the required operations.

The optimization module of AREMOS is shown as the third layer in Figure 6.1. It is an optimization software package, hence it is a distinctly separate part of the system. The object oriented management system prepares the mathematical programming model using the data stored in the objects and sends it to the optimization module to obtain the optimum results. After getting and refining the results, it presents them to the user.

The following three sections examine the user interface, the object oriented management system and the optimization module. It should be emphasised at this point that the user interface part is not a distinct module of AREMOS. It is fully managed by the object oriented management system. On the other hand, they are viewed as separate layers of the system and covered under separate topics in the following sections. This is primarily due to the fact that such a conceptual differentiation help assess the AREMOS system from the user's point of view and also the programmer's point of view. The visual interactive user interface part concerns the user's side and the object oriented management system part concerns the programmer's side of the AREMOS system.

6.1. The User Interface

As many other Windows applications, AREMOS appears on the screen as a big window frame with a menu bar. It is basically used through selecting menu options and clicking the visual objects on the screen by a mouse. The refinery models appear as multiple overlapping windows within the outer window frame of AREMOS. Only one of these windows is active at a time, but the user can easily shift from one window to the other, with this, the user is able to deal with more than one visual refinery model simultaneously.

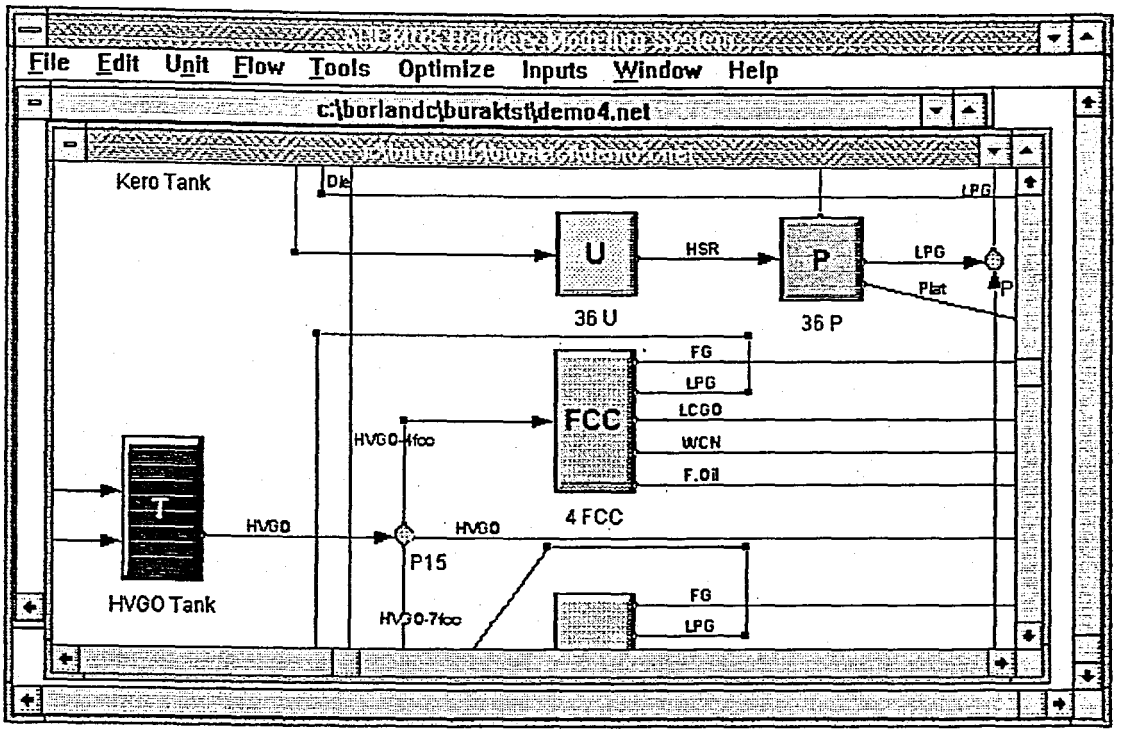


FIGURE 6.1.1 A View of the Sample TÜPRAŞ İzmit Refinery Model Developed in AREMOS

A refinery modeling window supports the construction of a network based representation of a refinery production scheme. This representation consists of graphic icons and graphic arrows standing for refinery process units and product flows respectively. Figure 6.1.1 illustrates a sample model generated for TÜPRAŞ İzmit Refinery. The whole modeling frame can be thought of as a big sheet that doesn't fit into a window, therefore only a portion of it is displayed at a time. The user can scroll through the model and reach any part of it via scroll-bars. A zooming utility is also provided to overlook the generated refinery scheme. The zoomed representation covering the entire model of TÜPRAŞ İzmit Refinery is depicted in Figure 6.1.2.

The menu bar provides model constructions tools, file management tools, and mathematical programming model generation and optimization tools. Besides these options that are central to refinery modeling process, an online help option and standard window handling options (ie. for tiling or cascading the refinery modeling windows) to facilitate screen management are supplied in the menu.

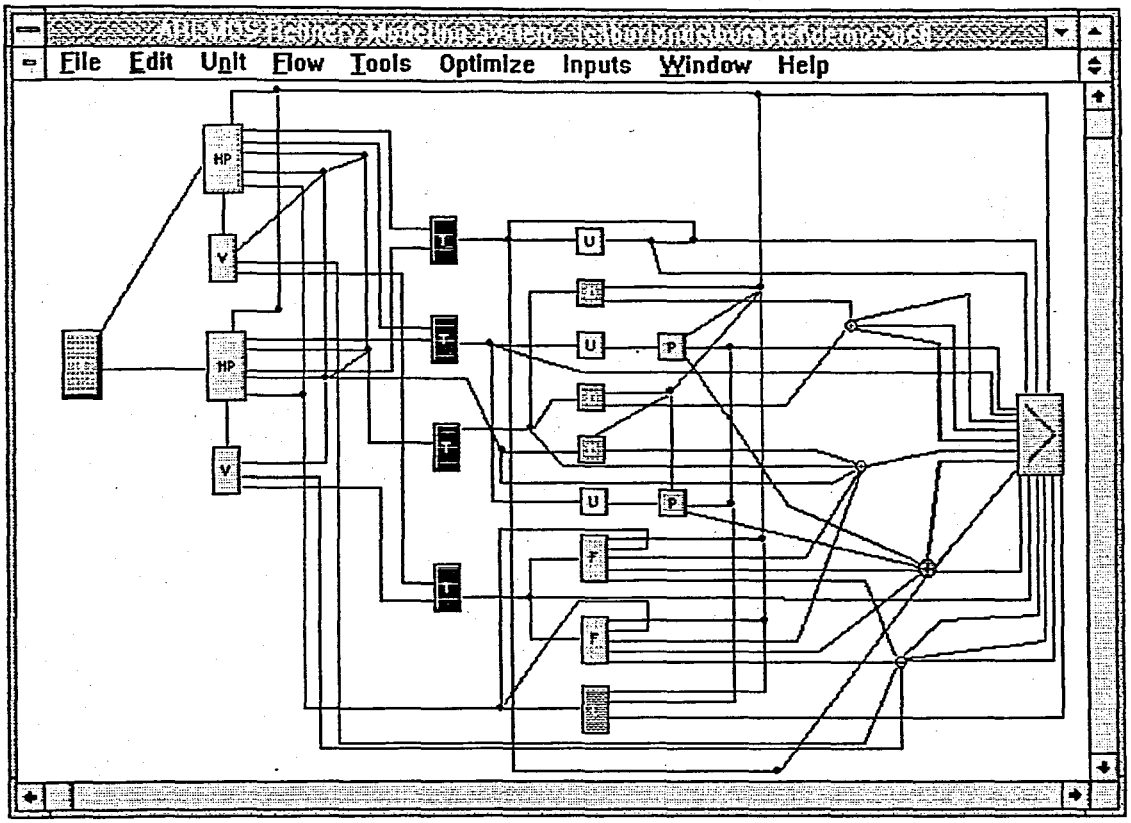


FIGURE 6.1.2 A Zoomed View of the Sample TÜPRAŞ İzmit Refinery Model in AREMOS

Most of the main menu options appearing in Figure 6.1.1 are used for opening pull-down sub-menus, thus represent categories of operations. 'File' menu option provides the file management utilities for refinery models. These utilities essentially enable the user to save a refinery model as a file, and to retrieve it later for further work. The pull-down menu options lying under the 'File' are namely 'New', 'Save', 'Save As', 'Close All', 'Delete' and 'Exit'. By selecting these, the user can generate a refinery modeling window, save the active refinery modeling window, save the active window with a new name, close all windows, delete a refinery model file, and exit AREMOS, respectively. In addition to the 'Close All' option, the user can also close a specific refinery modeling window by selecting the 'Close' option provided in the standard Control-menu box positioned at the upper left corner of the window.

The 'Edit' option has 'Delete', 'Copy' and 'Paste' as its pull-down menu options, however only 'Delete' is implemented for the time being. In order to delete a flow or a unit, the user first selects this option, then clicks on the flow to be deleted. When the 'Delete' is selected, the cursor changes to a cloud shape to indicate the deletion phase. This phase ends

when the user actually deletes an object or clicks an empty space on the screen. The 'Copy' and the 'Paste' can be implemented for carrying one or multiple screen objects from one network modeling window to another.

The 'Unit' option provides a long pull-down menu of all available refinery process unit types. The 'Flow' option, on the other hand, has no sub-menu and selected directly. The 'Unit' option together with the 'Flow' provide the means of developing a complete refinery production scheme on a refinery modeling window.

Refinery model construction starts with generating an empty refinery modeling window by selecting the 'New' from the 'File' sub-menu. Addition of a new refinery unit to the model is performed by selecting a unit type from the 'Unit' pull-down menu and clicking on the window. As a result, the corresponding unit object is created and displayed with its representative icon on the window. The place of a unit icon can be changed later by dragging it to a new position. If the unit has inflows or outflows, the associated flow arrows are automatically moved with the unit icon.

A product flow connection is generated by selecting the 'Flow' option and indicating first the outflowing and then the inflowing refinery units by clicking on their icons. Then the product flow appears as a straight arrow between the associated unit icons. Later the flow arrow can be given any convenient S-shape by clicking on a point on the arrow and dragging the point. This helps to generate a neater visual network format. Also a flow can be directed to a new unit by dragging its arrow head onto a new unit icon. In some cases, a flow connection might be invalid, then the flow can not be created and the user is warned about the error. As can be noticed, creating units and flows properly is enough to generate a visually complete refinery network.

The 'Tools' option has 'Zoom in' and 'Zoom out' options in its pull-down menu. The user can have a zoomed view of a refinery network by selecting the 'Zoom in' option, but no manipulation is allowed in the zoomed state. The user should select 'Zoom out' to turn to the normal view and continue the modeling process.

The 'Optimize' option provides the sub-menu for optimization utilities. These utilities are responsible for establishing the interface link between the visual refinery model, which is known to the user, and the associated mathematical model, which is managed behind the scenes by AREMOS.

'Compile', 'Run' and 'Report' are the menu commands provided under the 'Optimize' menu option. These commands can only be used after a refinery production scheme is completely developed. This requires that enough data be input to the refinery model, which is performed through dialog boxes as will be discussed later in this section. Nevertheless the 'Compile' command first checks the possible modeling errors and missing parts in the developed model. If the model is validated, the mathematical programming model associated with the visual refinery model is formulated and saved in a file. The 'Run'

command first compiles the model if it has not been compiled yet, then runs the LINDO optimization package and sends the mathematical model file to LINDO for obtaining the optimum production results. The 'Report' command takes the optimal solutions from LINDO and reports the results after a pre-processing. This is a simple report utility which states whether the mathematical programming model results in an infeasible solution, an unbounded solution or a feasible solution. If the solution is feasible, it reports the objective function value and the optimal values of the important variables which are the amounts of product flows between refinery units as well as the amounts of crude oils, component exports and final products. Figure 6.1.3 illustrates the optimal results reporting dialog box appearing after optimizing the sample model of the TÜPRAŞ İzmit Refinery.

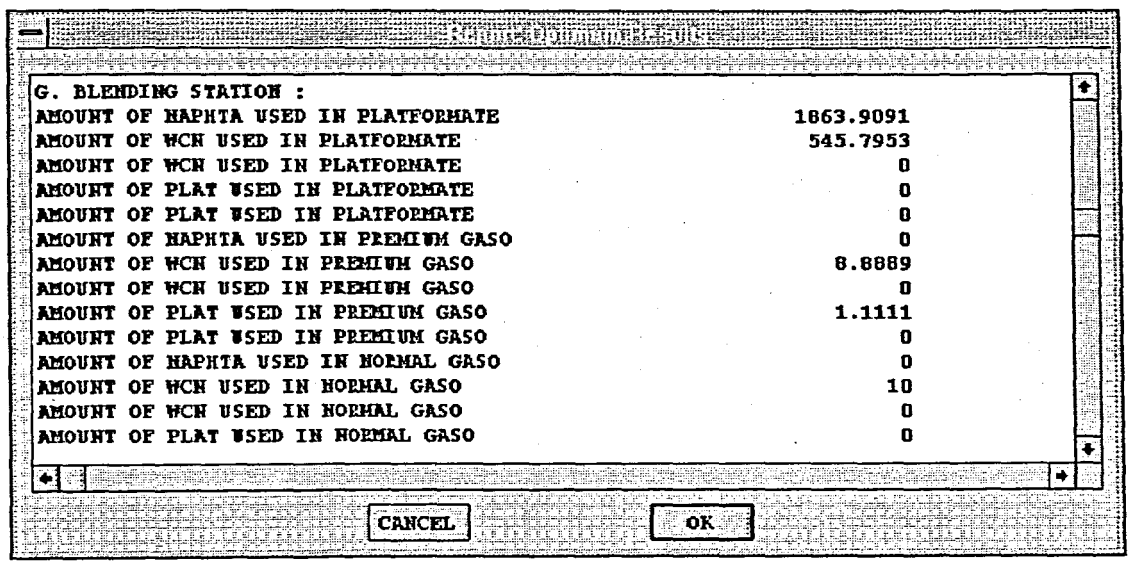


FIGURE 6.1.3 The Optimal Results Reporting Dialog Box for the Sample Refinery Model

As a typical property of Windows operating system, data exchange is realized through dialog boxes in AREMOS. Units and flows of a refinery contain the essential part of the data, therefore most of the refinery-specific data exchange is realized through dialog boxes that are related with the screen objects. The associated dialog boxes are opened by double-clicking on these network objects. Figures 6.1.4, 6.1.5 and 6.1.6 illustrate the dialog boxes of a gasoline blending unit, of an HP distillation unit, and of a product flow going into a blending operation. These units belong to the sample TÜPRAŞ İzmit Refinery model depicted in Figures 6.1.1 and 6.1.2.

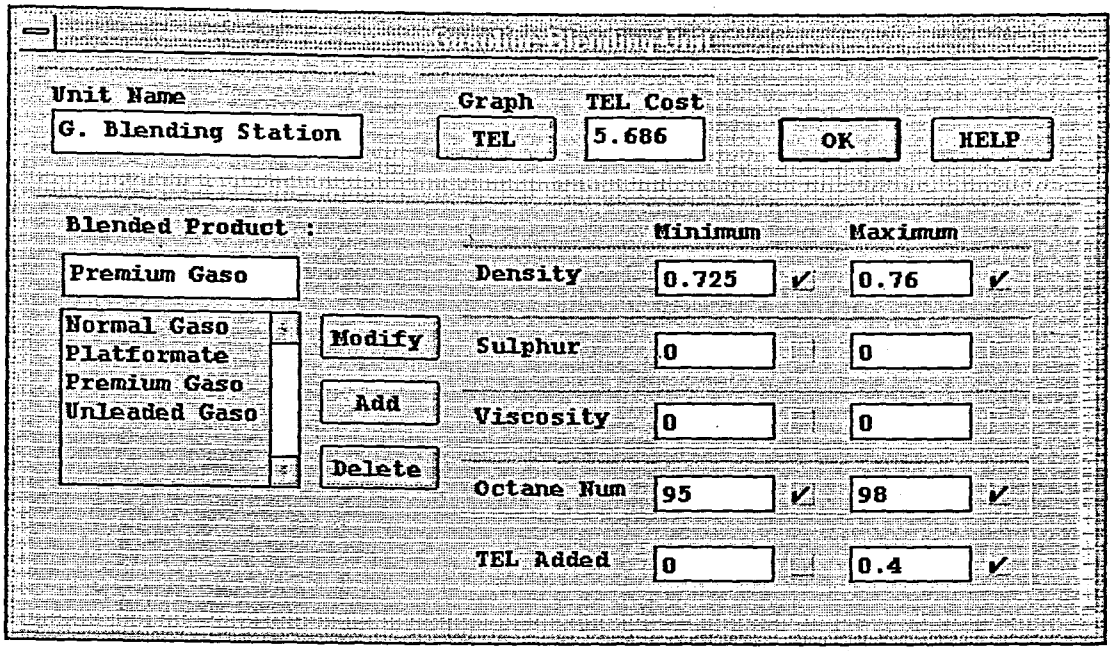


FIGURE 6.1.4 The Appearance of an Input Dialog Box for a Gasoline Blending Unit

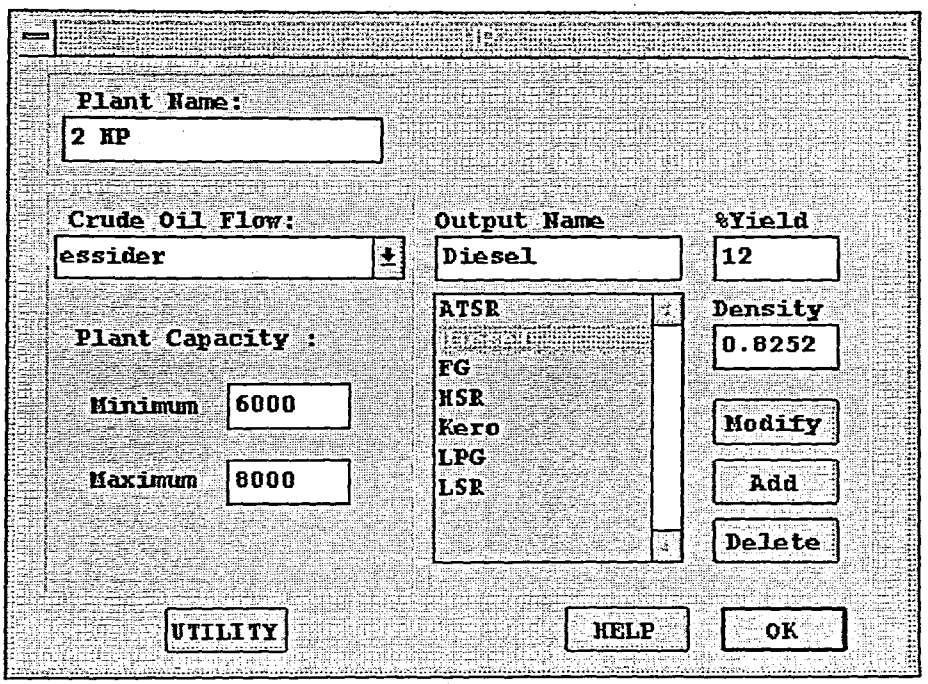


FIGURE 6.1.5 The Appearance of an Input Dialog Box for an HP Distillation Unit

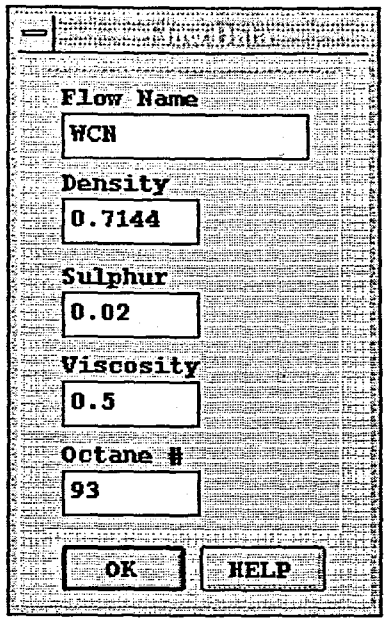


FIGURE 6.1.6 The Appearance of an Input Dialog Box for a Product Flow

All these dialog boxes have similar appearances, because they make use of the standard Windows data exchange resources such as edit boxes, list boxes, buttons, etc. Note that pressing the TEL, HELP, and UTILITY buttons in these dialog boxes results in opening of additional dialog boxes. For instance, Figure 6.1.7 illustrates the utility dialog box opened as a result of pressing the UTILITY button in the dialog box of the HP Distillation unit, which is given in Figure 6.1.5. Usage of such subsidiary dialog boxes through buttons are implemented in most dialog box designs.

The remaining menu options that are not covered yet are 'Input', 'Window' and 'Help'. Briefly the 'Input' provides sub-menu options for entering general data about the refinery production scheme. The 'Window' provides 'Cascade', 'Tile' and 'Arrange icons' options for rearranging the shapes of refinery modeling windows. Finally 'Help' provides general help on refinery modeling.

As a general principle, the interface of AREMOS guides the user throughout the modeling process. It checks the validity of the construction at any step, and communicates feedbacks given by the object oriented management system in case of any dangerous or infeasible construction attempt. An overview of the associated model construction rules will be given in the detailed explanation of unit and flow classes. For example an error message appears with necessary explanations if the user tries to create an invalid flow connection or enter invalid data in a dialog box. Also help buttons appear in almost all dialog boxes to

provide the user with on-time help about the operations related with the active dialog box (ie. the help button in the gasoline blending dialog box in Figure 6.1.4).

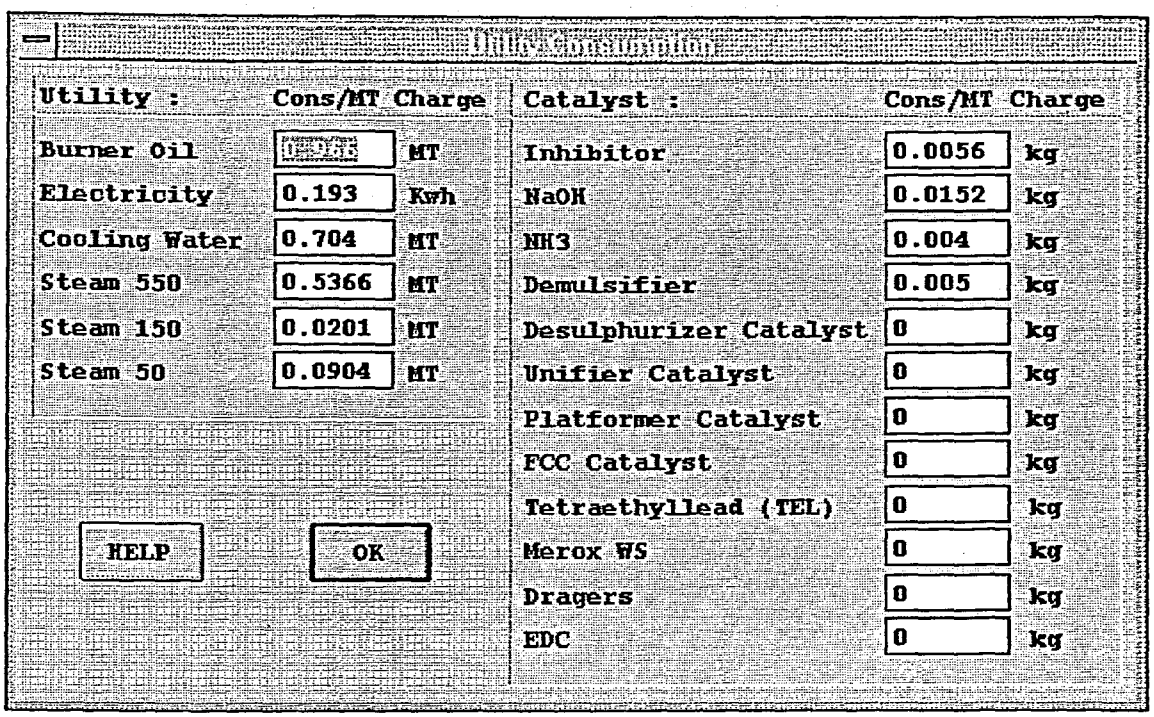


FIGURE 6.1.7 The Appearance of the Utility Dialog Box for an HP Distillation Unit

An important characteristic of AREMOS is that it virtually eliminates the need for mathematical programming expertise in model construction, rather encourages the user to use his/her refinery engineering knowledge. Furthermore its interface doesn't permit the user to see or intervene with the mathematical programming model. Such an interface is easily achievable in refinery modeling, because petrochemical operations can be directly modeled as mathematical relations formulated in the form of well defined functions. Consequently, AREMOS generates the linear programming model using the available data without any mathematical programming guidance of the user.

6.2. The Object Oriented Management System

The object oriented management system of AREMOS has three basic components: The interface management component, the database management component and the optimization management component. These components and their relations with the optimization module, the interface and the refinery network models are illustrated in Figure 6.2.1.

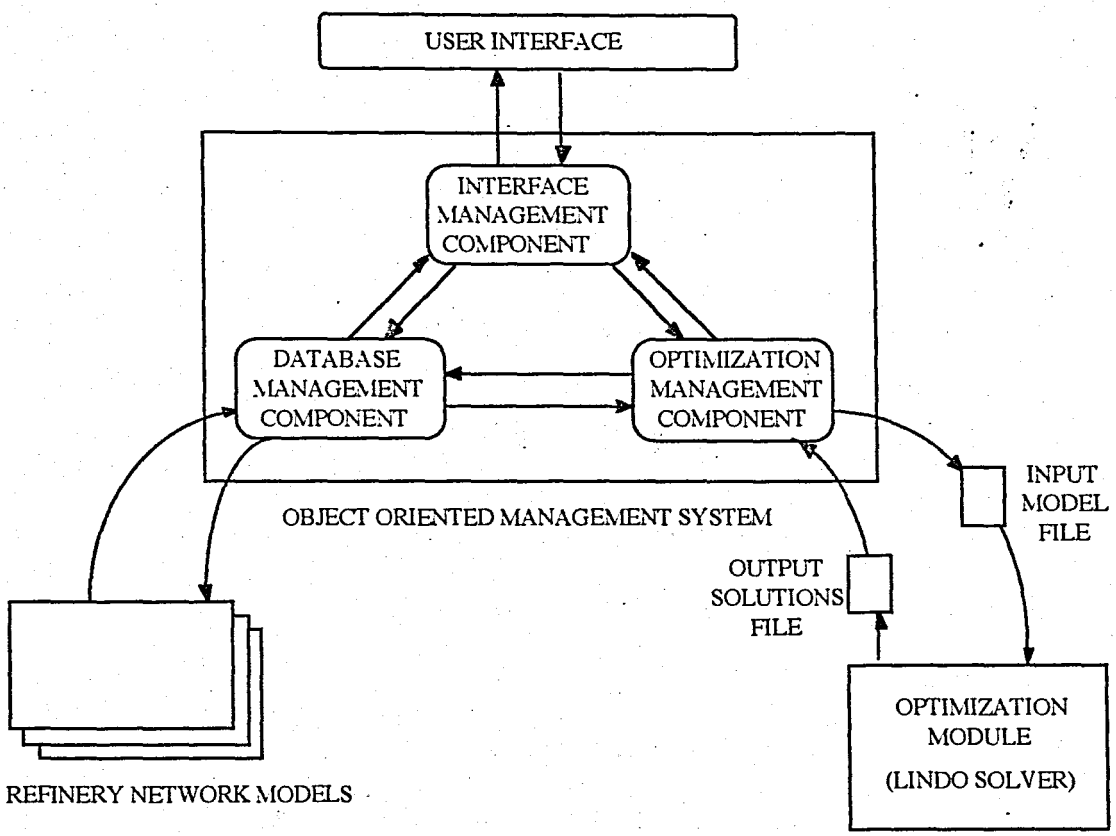


FIGURE 6.2.1 The Components of the Object Oriented Management System

The object oriented management system runs in an event driven way, that is, it runs according to the events produced by the user on the interface. Moving a refinery icon on the screen, for instance, is a user-driven event, and the interface component interprets such events and generates the related messages to be processed by the other components or by

itself. Turning back to the previous example, moving an icon requires the tasks of detecting the specific icon and its underlying refinery unit object, erasing the icon and drawing it in its new position, and updating the window coordinates of the refinery unit object. These require a series of message dispatches between interface management and database management components.

Database management and optimization management components have no direct connection with the interface, instead the interface management component processes every kind of interface event and sends the necessary messages to these components. Database management component is responsible for maintaining refinery models with respect to the user interface events. It also performs refinery model storage and retrieval operations. As discussed in the user interface section, it is able to maintain multiple refinery models simultaneously. Optimization management component has the responsibility of processing the data of a refinery model kept by the database management component and producing the corresponding mathematical programming model as a text file. It also establishes a link between these different model representations for further work. If errors are encountered in this process, interface management is invoked for the display of the generated error messages. Once the validation and construction of the mathematical programming model is successfully completed, the user can run this model. When a run command is given through the interface, the optimization management component sends the generated text file to the optimizer which is a distinct module of AREMOS. The optimizer is an LP solver package, which solves the linear programming model and presents the optimal solutions as a text file. Using the link with the visual interactive model, the optimization management component interprets the optimum results and prepares an easy-to-understand report file. Finally these results are presented to the user.

This component-wise view of the management system facilitates better understanding of the functioning of AREMOS. However, the system is designed using an object oriented approach, and object oriented systems put emphasis on the data, not on the functions. The functions to be performed by the system are designed as behaviors of objects, so that the data and the functions to be applied on the data are encapsulated in the objects. Therefore, the three basic functional components of the object oriented management system need further review in this respect.

In fact the design of the management system doesn't include any distinct components. Instead, the interface management, database management and optimization management functions are implemented as the methods of the refinery network objects, refinery unit objects and product flow objects. As mentioned in the previous sections, these three object types are central to the design of the management system, yet there are plenty of other object types (classes) used in the design. The class hierarchy, the data structures in

which the objects are maintained, the message passing rules, and the arrangement of relations with the operating system will be examined in detail in the succeeding sections.

6.3. Optimization Module

The optimization module of AREMOS integrates an optimization software package for solving the linear programming model associated with the visual interactive refinery model that is produced on the interface. However this study does not put emphasis on the selection of a specific software package, nor on the communication of the object oriented management system with such a package. Rather, what is needed is merely a linear programming solver for solving the refinery linear programming model and to obtain the optimal production results. The LINDO optimization package is used in AREMOS for this purpose, but this is an arbitrary selection. Later, it can be replaced with any other solver. In fact, ideally pre-written linear programming solver codes should be integrated into AREMOS, and such a replacement of LINDO is planned as a future improvement in AREMOS.

LINDO solves linear, integer, and quadratic programs, and accepts symbolic or MPS formatted input. It has a user interface which is integrated with a solver. It comes with a number of commands, which can be used interactively. However, its usage is restricted in AREMOS. It is used just as a linear programming solver. LINDO is very appropriate for this purpose since it requires almost no specific language knowledge. The optimization management system of AREMOS produces the linear programming model of the refinery as a symbolic formatted LINDO input file in which all constraints and objective function are written explicitly as algebraic expressions. However, no LINDO specific command or modeling method is used in the derivation of the mathematical model (except MAXIMIZE and SUBJECT TO commands), therefore mathematical program derivation methods of AREMOS can be easily adapted to any other optimization package. LINDO provides the utilities for batch runs and sensitivity analysis, yet these are not used in the prototype system.

The optimization algorithm in LINDO is a black box in the sense that once the user invokes the algorithm, it runs to completion. This is the common practice in many optimization packages for linear programming although some of them allow step by step execution of the algorithm. Nevertheless, such a step by step optimization is not the aim in

AREMOS, rather each execution should be regarded as a step in the modeling process. Therefore refinery modeling and optimization include multiple runs of the system. Each successive run aims to have better fit for the real refinery process, and for better objective function values.

7. DESIGN CHARACTERISTICS OF THE SYSTEM

7.1. Essential Characteristics of the Operating System

Microsoft Windows version 3.1 is the target operating system for the system development of AREMOS. It runs under IBM-compatible personal computers. The Windows operating system provides a graphic-based multitasking windowing environment that allows the programs written specifically for Windows to have a consistent appearance and command structure.

Windows has three major capabilities: a graphics-oriented user interface, hardware independence, and a multitasking capability. Individually, none of these capabilities is new itself, but attempting to combine all three of them into a single microcomputer operating environment is a new concept for operating systems.

Windows partitions the screen into rectangular windows. Each application program occupies a window on the screen, and can use many other windows for different purposes. Each window has a border and a client area, may have a title bar, a system menu, minimize-maximize boxes, a scroll bar, a menu bar. The client area of a window is the primary output area for a program. Windows provides several built-in routines that allow the easy implementation of pull-down menus, scroll bars, dialog boxes, icons and many other features of a user friendly graphical interface. These provide a standardized user interface for application programs running under Windows, which are also called Windows applications.

Another capability provided by Windows is hardware device independence. Windows liberates the programmer from having to account for every possible variety of monitor, printer, and input device available. As a result, the program developer interacts with Windows rather than with any specific device. Therefore it is Windows's responsibility to accomplish the tasks required from the programmer for connecting to existing devices.

Multitasking capability is the most important feature of Windows from the programmer's viewpoint. The multitasking operating environment allows the user to have several application programs, or instances of the same program, running concurrently.

Memory is an important shared resource under Windows. With more than one application program running at the same time, each program must cooperate to share memory in order not to exhaust the supply. Also, as new programs are started up and old

ones are terminated, memory can become fragmented. Windows is capable of consolidating free memory space by moving blocks of code and data in system memory.

Another important shared resource is input from the keyboard and mouse. It is for this reason that C++ language statements for direct input and output are not used for Windows programming. Windows specific function calls are used instead. Under Windows, an application does not make explicit calls to read from the keyboard or mouse input devices. Instead, Windows receives all input from the keyboard, mouse, and timer, in a system queue. It is the queue's job to redirect the input to the appropriate application program by copying it from the system queue into the program's queue. At this point, when the application program is ready to process any input, it reads from its queue and dispatches the right message to the correct window.

The principal means used to disseminate information in the multitasking environment is the Windows message system. In fact, Windows is a message-based operating system since all its functioning is realized through dispatching messages. From the program's point of view, a message can be seen as a notification that some event of interest has occurred that may or may not need a specific action. These events may be initiated on the part of the user, such as clicking or moving the mouse, changing the size of a window, or making a menu selection. Messages can also be generated by the application program or Windows itself. The principal effect of this processing style on the programmer's side is that the program written for Windows should be totally oriented toward the processing messages. A program must be capable of awakening, determining the appropriate action based on the type of message received, taking that action to completion, and returning to sleep.

In order to equip the programmer with the necessary tools to program under Windows, Windows provides an application program with access to about 450 function calls. Nevertheless programming under Windows requires a complete comprehension of Windows concepts, functions and message processing rules. Borland C++ compiler used for developing the AREMOS system provides ObjectWindows as an alternative way for developing Windows programs. Borland's ObjectWindows provides a powerful object-oriented library to the standard Windows programming environment. It is a complete collection of objects that describe standard Windows features. Appendix A gives detailed information about ObjectWindows.

7.2. Object Class Hierarchy

A well-known characteristic of object oriented programming is the inheritance process by which one object can acquire the properties of another object. If an object is derived from another object, the former is called the parent and the latter is called the child object. A child object inherits the data and methods of its parent, and has extra specific data and methods. With multiple inheritance, an object may have more than one parents and child. In this way additional specification objects can be derived from general objects and these objects can be hierarchically classified.

An object class hierarchy, which is also known as an 'inheritance tree', shows the overall parent-child relationships between the objects used in an object oriented design, and is very helpful for having a general view of the design.

The object class hierarchy tree of the developed system is depicted in Figure 7.2.1. A basic design characteristic that can be grasped at first sight from the figure is that all the relations between child and parent objects are of single inheritance, that is no child object has multiple parents. All object classes in the system derive from a base object class named 'Object'. Object is the base class for all Borland ObjectWindows derived classes and is defined in Borland Container class library. There are also other base classes that belong to these class libraries which have no importance in this design, and hence not shown in the hierarchy tree. Object, on the other hand, is essential to the design, and used as the root for all the classes generated for the development of the refinery modeling system. Having such a root class provides plenty of advantages as will be discussed later.

TWindowsObject, TWindow, TMDIFrame, TMDIClient, TModule, TApplication, TDialog, TFileDialog are ObjectWindows classes which are central to the design of the system. RefApp, RefMDIFrame and RefMDIClient are derived from these classes, and are responsible for the management of the overall system and establishing the connection with Windows. TDialog class serves as a base for derived classes that manage Windows dialog boxes. Derived dialog classes are associated with dialog resources, and have methods to handle communication between a dialog and its control objects. Within dialogs, controls such as list boxes, buttons, edit boxes, scroll bars allow users to enter data and select options.

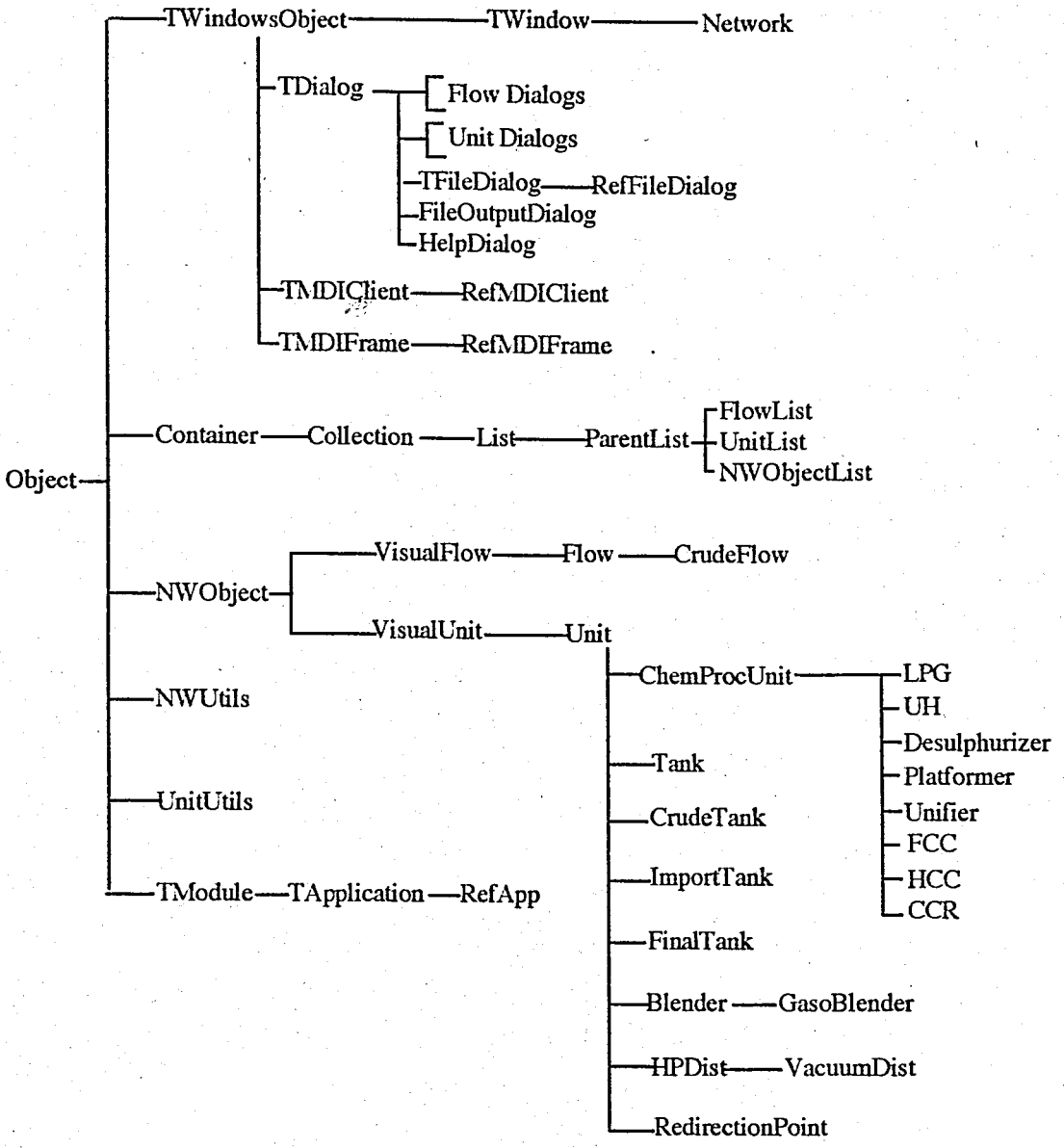


FIGURE 7.2.1 The Object Class Hierarchy of the AREMOS System

Control objects provide consistent and simple means of dealing with all kinds of standard controls defined by Windows operating system. Control object classes derive from TDialog, and are frequently employed in dialog designs of the refinery modeling system. Many TDialog derived classes are used in the design, however only the help dialog, the refinery file management dialog, and the file output dialogs are shown explicitly in Figure 7.2.1. In addition to these, 'Flow Dialogs' and 'Unit Dialogs' in the class hierarchy refer to two groups of derived dialog classes which are used in conjunction with refinery unit and flow classes. Borland Container class library provides many data structures in the form of

object classes. List, DoubleList (doubly linked list), Array and BTree (binary tree) classes are just a few of them. Only List from this library is employed in the system design of AREMOS. It is used as the parent for the list classes of the system. List object requires that all the objects that will be kept in the list should have Object as their parent object. The benefit of this is that it can contain heterogenous objects, that is objects from different classes, provided that they have Object in their parental roots. This condition is satisfied by every object in the object class hierarchy of the AREMOS system.

Borland Container class library provides many data structures in the form of object classes. List, DoubleList (doubly linked list), Array and BTree (binary tree) classes are just a few of them. Only List from this library is employed in the system design of AREMOS. It is used as the parent for the list classes of the system. List object requires that all the objects that will be kept in the list should have Object as their parent object. The benefit of this is that it can contain heterogenous objects, that is objects from different classes, provided that they have Object in their parental roots. This condition is satisfied by every object in the object class hierarchy of the AREMOS system.

Network derives from TWindow which is a general purpose window class whose instances (objects) can represent different types of windows (ie. main, pop-up, child windows) of a Windows application. Network is the underlying object of a visual refinery modeling window. Network inherits its windowing characteristics from TWindow and adds refinery data and methods. The basic duty of a Network object is the maintenance of flow objects and unit objects and their interface connections. These unit and flow objects appear as icons and arrows on the refinery modeling window.

NWObject inherits its properties from Object, and is the parent object class of all unit and flow classes. VisualFlow and VisualUnit derive from NWObject, and supply the visual properties to their offspring which are Unit and Flow classes. Unit object class models the general characteristics of a refinery processing unit and serves as the parent of all refinery processing unit classes. A sub-category of refinery units is ChemProcUnit. ChemProcUnit possess the general modeling characteristics of the chemical processing units. The most significant of these characteristics is the usage of input-output yield tables for mathematical modeling. Flow object class models the refinery product flows, and is the parent of CrudeFlow class. CrudeFlow utilizes the visual properties of Flow, however represents not only one flow but multiple crude oil flows.

Many of the refinery units have variable running costs which depend on the amount of unit input charge. All the cost items used in the refinery, which are called unit utilities, are modeled in UnitUtils class. A unit object with utility consumption holds an embedded UnitUtils object for keeping its utility data. NWUtils is very similar to UnitUtils and used by Network to keep general information on the utilities.

There are also other supplementary classes not appearing in the class hierarchy. Many of the objects make use of dialog box objects derived from TDialog for interface data exchange. Utility dialog boxes as well as flow and unit dialog boxes are among them. Moreover, the Network object, some unit objects and CrudeFlow object use internal objects for keeping their data. CrudeOil used in an CrudeTank unit object and CrudeOilYields used in an HPDist unit object are good examples of such internal objects.

7.3. Design of the Object Classes

This section aims to give the design of the object classes in terms of their data and methods. Some classes are considered under groups and some examined one by one as needed. In any case, data and methods are not explained explicitly, instead a general view of them is given with respect to the important design characteristics. On the other hand, just explaining the design of classes in this way is not enough to describe how the system maintains the objects at run time. Therefore this topic is separately examined in the next section.

Network, Unit and Flow classes are the central classes of the AREMOS system design, and the three management components of the object oriented management system are implemented through these classes. Therefore each of these classes have interface management duties, database management duties and optimization management duties which are covered in the following associated sections.

An important point to emphasize before describing detailed class designs is that C++ language, with which the system is written, has its own means for implementing the principles of object oriented programming. In C++, the methods of the objects are declared as functions, and message passing process between objects is performed through calls of object functions, which are also called the member functions. Suppose that A is an object of a system. In order to receive messages from other objects, A declares some of its member functions as public, which become accessible by other objects. By doing so, A sets a message receiving protocol, and other objects can only use this protocol to send a message to A, thus the inside of A is encapsulated from the outside. If object B of the same system wants to send a message to A, it can call one of the public member functions of A. An object can also have hidden functions for its internal usage. Note that the terms 'method' and 'function' are used interchangeably in the following object class descriptions.

Another point that may lead to confusion is the difference between the terms 'object' and 'class'. An object refers to an instance of a class, for example Kerosene can be an instance of Flow class. It is the objects that exist, or 'live', during the execution of a system.

The principle of polymorphism is implemented in C++ by means of virtual functions. A virtual function has a definition in a parent class, but its children may redefine it differently for their own purposes. Therefore a virtual function is a common interface to similar group of tasks in child classes. This encourages the programmer to define generic virtual function such as 'draw', 'write', 'erase'. Each child has its own implementation for these generic messages.

The term 'abstract class' is frequently used in the following design descriptions, and refers to an important concept in object oriented programming. Some classes cannot be instantiated in C++. These are called abstract classes which serve as an umbrella for related classes. As such, an abstract class has few if any data members, and some or all of its member functions are pure virtual functions. Pure virtual functions serve as placeholders for functions of the same name and signature intended to be defined eventually in derived classes. Abstract classes also have member functions that have certain definitions Children classes can use these functions directly, or can redefine and then use them. Abstract classes help the programmer to generate a more robust and reusable class classification, and make it much easier to extend and modify an object oriented system.

The class hierarchy of AREMOS includes Object, NWObject, ParentList, VisualUnit, VisualFlow, Unit, and ChemProcUnit as the abstract classes of the system and no instances of these classes exist at run time. They contain the data and abstract methods common to all child classes. Moreover, classes belonging to the two Borland class libraries are not instantiated at the system run time. Rather, they are treated as abstract classes and instances of their derivatives are made and used throughout the system run time.

The following is the description of the data and methods of the classes that appear in the class hierarchy diagram. Although the classes are examined under separate headings, cross-references among class descriptions are frequently needed, because it becomes difficult to examine a class without knowing its relations with its parents and children, and with other classes.

7.3.1. Object Class

At the top of the class hierarchy is the Object which is the abstract base class for all the classes in the system as well as for the Container class library and the ObjectWindows class library. Keeping such a common parent class is very useful in manipulating objects

through pointers. In C++ language, a pointer type to a parent class can be successfully used for pointing child classes. Therefore operations on child classes can be performed through parent pointers without any need to know the actual child class of the object which is being pointed to. This, of course, necessitates deliberate definitions of object methods.

Since Object is the base class for all classes, an Object pointer can handle any class of object used in AREMOS. A significant benefit of this is the ability of the list classes to keep different classes of object, ie. different unit objects such as an HPDist object and an FCC unit object.

Object contains one pointer data type for error handling, and pure virtual functions that provide the basic essentials for all derived classes in the Container class library including List class. 'isA' and 'nameOf ' are the names of two such functions which are frequently used in the system design. They return a unique number and a unique name to identify the specific child class of Object.

7.3.2. RefApp, RefMDIFrame and RefMDIClient Classes

AREMOS maintains one instance from each of RefApp, RefMDIFrame and RefMDIClient classes throughout its life cycle. The RefApp object establishes the connection of AREMOS with the Windows operating system. The RefMDIFrame and RefMDIClient objects work under the RefApp object, and are mainly responsible for the overall interface management of the AREMOS system.

The first requirement of an ObjectWindows application is the definition of an application class derived from TApplication class, which is derived from TModule. TModule defines behavior shared by both Windows library and application modules. TModule member functions provide support for window memory management and error-processing. TApplication class supplies the basic behavior required of all Windows applications.

RefApp is derived from TApplication as the application class of AREMOS. It inherits the behavior of creating and displaying the application's main window, initializing the first instance of the application for any task, processing the Windows messages the application will receive, and closing the application.

RefMDIFrame and RefMDIClient are derived from TMDIFrame and TMDIClient, which are the multiple document interface (MDI) classes of ObjectWindows. MDI is an interface standard for Windows applications that allow the user to simultaneously work with many open documents. In this sense, refinery networks are the documents of AREMOS.

The user can have many open refinery network windows, simultaneously model different refinery schemes on these windows.

AREMOS constructs one RefMDIFrame object and one RefMDIClient object. The RefMDIFrame object appears as the main window of AREMOS, which is hold by the RefApp object. The member functions of the RefMDIFrame object are concerned mainly with construction and management of MDI child window objects, that are the Network objects each has a window representation on the screen. They also manage the RefMDIClient object and menu selections. The RefMDIFrame object keeps Network objects in a hidden list and provides the basic member functions to manage them. On the other hand, the primary role of the RefMDIClient object is behind-the-scenes management of Network objects.

RefMDIFrame and RefMDIClient bring in a great modeling ease by providing the programmer with an interface for MDI program development. They manage the burdensome part of the Windows-related operations, and the programmer only deals with the essential body of the program without bothering the rest.

7.3.3. Dialog Classes

The dialog classes of AREMOS are designed for interface management. They can be regarded as the interface classes of the AREMOS system. They define the behavior of dialog boxes through which most of the data exchange between the user and the system is realized.

A dialog is an interface element whose creation attributes are specified in a Windows resource file, which is prepared separately from the C++ source code. The creation attributes in the dialog's resource definition define the appearance and placement of the dialog and its controls such as buttons, list boxes and edit boxes. However the resource definition does not specify the behavior of the dialog, instead it is described in the C++ code.

Dialogs are used as child windows within a main window to perform specific input-output tasks. There are two design options regarding the implementation of dialogs in an object oriented fashion. As the first option, all the objects take on the control of their own dialog communications. However this requires an object to dispatch Windows dialog messages to a dialog, and receive the user messages from the dialog. Consequently this way is burdensome and lengthy.

The other design alternative makes use of the TDialog class and control classes (ie. ListBox, EditBox, Button classes) of ObjectWindows. TDialog class serves as a base for derived classes that manage Windows dialog boxes. Its member functions enable the

programmer to handle communication between a dialog and its controls. Moreover usage of ObjectWindows control classes make it much easier to display and retrieve data in an object oriented fashion. To take advantage of these features, this alternative is chosen in the dialog designs of AREMOS.

A drawback of this choice is that it necessitates the generation of several dialog box classes. Nevertheless the resulting modeling ease compensates this disadvantage. The dialog classes are designed in such a way that a dialog class serves either to a class or to a group of related classes. For example a dialog class is derived from TDialog for each of Tank, CrudeTank, FinalTank, Blender, GasoBlender, Flow, and CrudeFlow classes. On the other hand HPDist and VacuumDist use the same dialog class generated for distillation process, and similarly all the child classes of ChemProcUnit shares one dialog class. There are also other complementary dialog classes which are invoked by button presses or menu option calls. These are used to input data to a specific part of an object, which can itself be an internal object. The UnitUtils dialog object, for instance, is used to input utility data into a refinery unit object, and invoked by a button placed on the main dialog box of the unit object.

Unit and flow dialogs constitute the main body of dialog process in AREMOS. When a unit or flow is double-clicked on the screen, the corresponding dialog object is created, and the dialog box appears on the screen. The dialog object has the responsibility of verifying the user entries. Once the data exchange is over, the dialog box disappears and the dialog object is deleted.

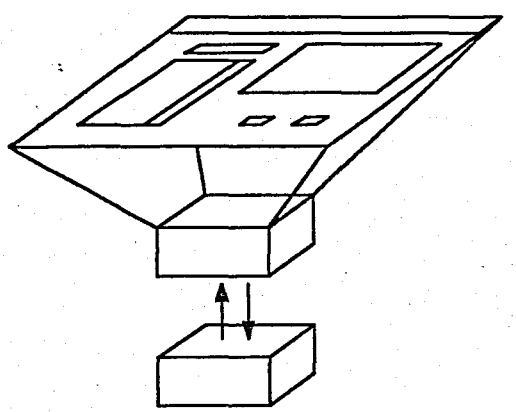


FIGURE 7.3.1 Object-Dialog Object Association during Data Exchange

RefFileDlg, HelpDlg and FileOutputDlg dialog classes are designed to fulfil some general tasks, rather than establishing a connection with an object's data. RefFileDlg is invoked through File menu commands. It is derived from TFileDlg that allows the user to choose a file for any purpose. RefFileDlg defines these tasks as opening, saving, replacing, and deleting a network modeling file. FileOutputDlg is a general purpose dialog class which simply outputs the contents of a specified disk file in a large dialog box. This is especially helpful in outputting the optimal results after solving the mathematical programming model of a visual refinery scheme. HelpDlg is designed for general usage. Help buttons are provided in unit and flow dialog boxes, and a help option is also provided in the main menu bar for general help on network modeling. When the user requests help through these selections, a HelpDlg object is created and the name of the specific help file is passed to the dialog object. HelpDlg object provides a list box and a multi-line editing box on the interface dialog window. Reading the help file, it fills the available help subtopics in the list box and demonstrates the corresponding explanation in the editing box as the user requests.

It can be concluded that the dialog objects has a key importance in the design of the overall system, since first of all, they acts as the interface part of the several objects, and have many data and functions for visual operations. If dialog objects were not generated all these functions would be embedded in the real objects of the system. Second, they define the ways how the data are represented and retrieved; and this determines the effectiveness of the visual interactive interface to a great extent.

7.3.4. List Classes

List classes of the AREMOS system, which derive from the ObjectWindows's List class, are designed for performing database management tasks. A refinery network object maintains its unit and flow objects, which hold the main data of a refinery network model, in two different list objects. Also, a unit object keeps list objects for maintaining its inflows and outflows. Section 7.5 can be referred to see illustrations which depict how unit and flow objects are maintained in list objects.

ObjectWindows's List class implements a linear, linked list. Lists are unordered collections in which objects are linked in one direction only to form a chain. Objects are added at the start of a list, but any object can be removed from a list. ObjectWindows also provides a list iterator class to traverse a list from head to tail to access the objects sequentially. By using member functions of the List class, any basic list operation can be performed easily. They release the programmer from the complicated pointer operations which becomes necessary otherwise.

A big advantage of List is that it can contain objects from different classes provided that they all derive from the base Object class. This is the case in the system design of AREMOS. This feature facilitates the object oriented design of the system as will be explained while describing the Network class. Another important feature is that an object contained in a List object can also be contained in other List objects, that is, it can belong to more than one List object at the same time. This type of data management is typically expected in an advanced data structure library like Container class library, and is inevitably needed in maintenance of objects in AREMOS.

ParentList derives from List and has added functionality for list operations. It serves as the base class for UnitList, FlowList and NWObjectList whose instances are produced and extensively used in the system. ParentList objects are also implemented in the internal designs of some unit classes to hold internal objects.

UnitList keeps a heterogenous list of refinery unit objects. It has no data part, and its member functions are aimed to facilitate visual operations such as drawing all units or searching for a unit at given coordinates. Visual operations are intensively used in the system, and such list functions reduce the burden of the programmer. FlowList is very similar to UnitList, and has functions for visual operations to be performed on the flow objects. Refinery network object has a FlowList object and a UnitList object to maintain all units and flows used in the refinery model. Another place where FlowList objects are used is unit objects. A unit object maintains two FlowList objects to maintain its input flows and output flows.

NWObjectList has a totally different function. As can be understood from its name, NWObjectList is generated to maintain the objects from the classes having NWObject in their roots. These include all classes of units and flows. NWObjectList is used by Network objects. Each Network object holds one instance of NWObjectList, and stores a representative object from each unit class and flow class in it. The main purpose of NWObjectList is to localize the non-object oriented features of the Network class. This is explained in detail while examining the Network class. It has also some other usage which is covered under the topic of unit and flow classes.

Storing units and flows in lists enables the generation of unlimited number of units and objects, therefore the free memory of the computer imposes an upper bound on the total number. Otherwise, arrays would be used for this storage, and the predefined size of the arrays would be an artificial bound for the number of stored objects. On the other hand, lists keep their members in an unordered sequence which is a handicap for the tasks which need to be performed with respect to some order of list members. Nevertheless, no such ordered performance of tasks on units and flows are needed in the system design of AREMOS.

7.3.5. Network Class

Network objects implement the three functional components of the object oriented management system. The interface management, database management and optimization management functions of the system are implemented as the methods of the refinery network objects. Refinery unit objects and product flow objects also have the same categories of methods, which are used by the methods of the refinery network objects.

The Network class models the visual environment for modeling a refinery in terms of its product flows and process units. A Network object is represented as a window on the screen, and normally contains a network of visual flows and units. The RefMDIApp object can maintain several such Network objects, which enable the user to model multiple refinery schemes simultaneously.

Network is derived from ObjectWindows's TWindow class. TWindow defines most of the fundamental behavior of a Windows's window, that includes the behavior for opening, closing, painting, and scrolling windows. TWindow also includes the behavior for command message processing and child window management.

Since Network has TWindow as its parent, it has the graphic functions of TWindow. Upon these functions, Network adds new data and methods for refinery modeling. Network provides several member functions for interface management, database management and optimization management. The data part comprises a relatively small number of data elements. The main data elements are a UnitList object, a FlowList object and a NWObjectList object. UnitList and FlowList objects keep the flows and units of the refinery network. A Network object manages all its units and flows by means of these lists. Implementation of the NWObjectList will be discussed later in this section. There are also many supplementary pointers and boolean (binary) data elements defined to carry temporary information among the member functions of the Network. These can be considered as internal message carriers.

In essence, object oriented programs perform computations by passing messages between active objects. In the system design of AREMOS, Network objects have the responsibility of determining active objects and passing messages to them. The objects of the system that can be activated by a Network object are all unit objects, flow objects and others such as utility objects. As it is the case in many windows applications, it is the user who actually directs the system and initiates the objects in AREMOS. A Network object receives user messages through mouse inputs or menu selections, and interprets them. Then it determines which objects are to be activated and which messages will be passed to these objects. In this sense, Network is in charge of handling the redirection of messages from the

user. For instance, when the user double-clicks an HPDist icon on the screen, the Network object receives the double-click message and screen coordinates, makes a search in its unit list and flow list to check if any object is at that coordinates. After detecting the HPDist unit object, the Network object passes a 'Receive-input-through-dialog' message to it. After this point, it is the HPDist object's responsibility to perform the necessary dialog operation. Once this operation is over, the control passes back to the Network object. An active flow or unit object can send messages to their Network object, or send messages directly to other objects. A flow object, for instance, can communicate with its head and tail unit objects. In any case, when the request of the user is performed, the Network object takes the control back.

In order to have a healthy object oriented design, the Network class is designed in such a way that among all unit and flow classes, only Unit and Flow are known to it. In other words, a Network object cannot distinguish between, for example, an HPDist and an FCC object, nor between a Flow and a CrudeFlow object. Therefore new unit classes or flow classes can be derived from Unit and Flow classes respectively (or from any descendant), and can be easily plugged into the system. Consequently, unit and flow classes are designed according to this design consideration. They make use of the principle of polymorphism, and provide functions that respond differently to the same messages. For example, an abstract 'Receive-input-through-dialog' function is defined in Unit class, and each child unit class redefines it for its specific dialog communication.

A detailed view of the object oriented design of the Network class can be given by examining its function types. As explained previously, the functions of a Network object can be classified as interface management, database management and optimization management functions, by means of which the three functional components of the object oriented management system are implemented.

7.3.5.1. Interface Management Functions. Network interface management functions establish the connection between the underlying objects such as units and flows, and the user interface. They receive any kind of user message from the interface, interpret the message, and dispatch it to the necessary place. The database management and optimization management functions are deliberately separated from interface management functions in order to have a more consistent design. Any kind of user message, a mouse-click or a file menu command for example, is first processed by a distinct interface management function, then the necessary functions are called. Therefore the database management and the optimization management functions do not have a direct connection with the user interface. Network interface functions also have the responsibility of displaying the visual result of an operation, if there is any.

The variety of operations that can be performed via the interface are explained in detail in the section reserved for the user interface. For any kind of menu or mouse operation, there is a corresponding interface management function. Some user interface operations only require interface functions. For instance, when the user changes the shape of a flow object, this operation results in calling a series of interface management functions which communicate with the flow object. However, if the user erases or adds a new flow, the interface management functions also call database management functions to delete or add the flow object to the flow list. Optimization operations, on the other hand, are only performed through menu commands. Consequently, there are interface menu functions specifically designed for optimization. When called, these functions invoke necessary optimization management functions. For instance, when the user selects the 'Compile' menu item, the related 'Compile' interface function is activated, and this function calls the optimization management functions for model validation and mathematical model formulation.

7.3.5.2. Database Management Functions. The database management functions perform the maintenance of the objects such as addition and deletion of units and flows, but they don't interfere in the internal data management of any object. Instead, each object manages its own data. In this respect, the database management functions work on the global data which are mainly the units and flows.

A subgroup of database management functions deal with file retrieval and storage operations which are performed through menu selections. Each unit and flow object has its own saving and retrieving functions. A Network object is saved as a file by calling the saving functions of each object one by one. Finally supplementary objects such as NWUtils are saved in the network file. The retrieval operation is the reserve of this operation.

7.3.5.3. Optimization Management Functions. The optimization operations are similar to the file management operations. They are performed through optimization menu commands. The optimization management functions mainly perform the tasks of validating the refinery network model, organizing the preparation the mathematical model, running the model and reporting the results.

Validation and preparation of the mathematical model are performed by calling the validation and mathematical program generation functions of the unit objects. Each unit object produces its mathematical program code and writes them into a disk file specified by the Network object. The flow objects, on the other hand, do not produce any mathematical programming code, but provide supplementary data to their head and tail unit objects for this operation. Appendix C gives the mathematical programming model generated by AREMOS

for the representative refinery model developed for TÜPRAŞ İzmit Refinery that is depicted in Figure 6.1.1.

Once the mathematical program file is generated, a temporary ParentList object, called 'VariableList', is formed, and each unit object is requested to store the meanings of the generated variables in this list. The objects stored in the VariableList is of type 'VarExp' (variable explanation) which is derived from Object class. VarExp is a supplementary object class with 4 data fields:

(a) Variable name produced by a unit;
(b) Meaning of the variable that is known to the user;
(c) Name of the unit object that produces the variable,
(d) Name of the head unit if the variable corresponds to an explicit product flow.

Appendix D gives a list containing the first three fields of these objects that are produced for the representative refinery model for TÜPRAŞ İzmit Refinery.

In the running phase, the mathematical program file, the one given in Appendix C for instance, is used as the input file to the solver, and the solver is executed. After the optimal solutions are received from the solver, the Network report functions refine the raw results, generate their interpretations through the variable explanation list, and display them as a report. The final report presented to the user after optimizing the representative TÜPRAŞ İzmit Refinery model is given in Appendix E.

7.3.5.4. Non-object Oriented Features. In practice, it is very difficult to generate a thorough object oriented design. In the design of AREMOS, there are some situations, in which a Network object needs to know the exact class of a unit or a flow object. Nevertheless, such situations rarely arise, and localizing these non-object oriented features seems to be a good solution. NWObjectList object is utilized for this localization process. A representative object from each unit class and each flow class is stored in the NWObjectList object in the initialization phase of a Network object. By keeping such a list, a Network object knows the exact classes of objects it maintains, but indirectly. As a result of this localization process, the rest of the program body maintains its object oriented nature. Of course, this design solution requires special care. File retrieval is a typical operation for which the NWObjectList object is made use of. All the unit and flow objects of a Network object are stored sequentially in one disk file, and different classes of objects are distinguished by reading a unique code number in the retrieval operation. Retrieval of these objects from the file requires that the Network object should be able to understand these numbers, in other words, it should know every unit and flow class. Instead of doing the retrieval this way, the Network object reads the code number and searches the list for a match. When the object

with the same code number is found, the Network object orders it to retrieve the object data from the disk. Therefore the Network object has the representative object in the NWObjectList perform the retrieval operation. This is a longer process, but necessary for the safety of the object oriented design.

Another advantage of the NWObjectList object is that it enables the Network object to accomplish tasks which are related with the instances of a specific class in an object oriented manner. A typical example for these tasks is counting the instances of a class in the system. It has another advantage in handling the visual resources of flows and units. These aspects of the NWObjectList are discussed in detail under the topic of unit and flow classes.

7.3.6. NWObject, VisualUnit and VisualFlow Classes

NWObject, VisualUnit and VisualFlow classes are abstract classes which are designed to define the interface data and functions of Unit and Flow classes.

NWObject is an abstract class derived from Object. It serves as a base for VisualUnit and VisualFlow classes. NWObject has the data to handle the visual resources of Windows used by unit and flow objects. These resources include pens, cursors, brushes, bitmaps and fonts. In addition to visual resources, it has a data field which is utilized by the child objects for keeping the flow and unit names given by the user. NWObject also provides pure virtual functions for performing visual operations.

VisualUnit and VisualFlow define the visual properties of a unit object and a flow object respectively. They redefine the pure virtual functions of the NWObject, and have added data and functions for managing visual operations. VisualUnit controls the appearance and movement of a unit on the refinery network window. It defines the response functions to the user's mouse clickings on the unit icon. Likewise VisualFlow controls the presentation of a flow on the screen, however it has more complicated data structures and functions in comparison with VisualUnit, since there is a wide range of visual operations the user can do with a flow on the screen. VisualUnit and VisualFlow strictly define the behavior of visual units and flows, but are still abstract, because they let the child classes define the shape of a unit icon, and thickness and color of a flow arrow, and some other visual details.

Data input and output through dialog are not considered as visual operations, and VisualUnit and VisualFlow do not provide any dialog facility. Instead, each unit or flow object sets its dialog connection through its dialog object counterpart. This is explained under the heading of dialog classes.

Unit and Flow classes inherit their visual features from VisualUnit and VisualFlow. In other words, visual features of Unit and Flow are isolated in VisualUnit and VisualFlow classes. The reason for this isolation is that the visual features have a degree of dependence on the specific visual functions and data of the operating system. As a result of this design style, operating system-dependent visual functioning of the unit and flow classes is separated from the operating system-independent data and functions of the child classes. When an update of the classes are needed for another operating system, this design choice is expected to facilitate the maintenance process.

7.3.7. Unit and Flow Classes

Unit and flow classes have interface, database and optimization functions which work in connection with the respective management functions of a Network class.

Unit is the base class for all the derived unit classes, and Flow is the base for CrudeFlow. Unit and Flow take their interface features from their parents, and add functions for database and optimization management. Figure 7.3.2 illustrates this relationship.

Unit is an abstract class whose instances can not be produced directly. It provides the data which are common to all unit classes, and the functions most of which are pure and to be redefined by the child classes. On the other hand, the Flow class models a typical product flow in the refinery, hence it can be instantiated. A product flow, by definition, flows from one unit to another, hence a flow object defines a connection between two different unit objects. Generation of a flow requires the user to specify these units on the screen. Only after they are specified a flow can be generated. Throughout the life cycle of a flow object the tail unit remains the same, but the head unit can be changed by the user. The reason for this is that a product flow is produced by its tail unit, and it can be sent to a number of different units as it is the case in the real refinery process.

The main data of the Unit class are two ParentList objects, named InputList and OutputList, which hold the input flows and output flows of a unit object, and a UnitUtils pointer. The child classes who have utility consumption use this pointer to generate a UnitUtils object which maintains the utility data of the unit. There are also minimum and maximum capacity fields which hold the daily processing capacities of a refinery unit. The minimum capacity refers to the minimum feasible production level of a unit with respect to economical considerations, and the maximum capacity is the maximum physical production capacity.

The Flow class has simpler data fields as compared with the Unit class. Its important data are composed of density, viscosity index, sulphur content, octane number fields as well

as price, minimum demand, maximum demand, minimum supply and maximum supply fields which are utilized by import products and final products. It also holds two pointers for its head and tail units (outflowing and inflowing units).

The functions of Network class are grouped as interface management, database management and optimization management functions in the section reserved for the Network class. It is noteworthy that a network and its flows and units have matching classes of functions as can be observed in Figure 7.3.2. At this point, it will be helpful to examine these function categories of Unit and Flow classes separately. Later in this section, the communication between the functions of a network and the functions of its units and flows will be explained.

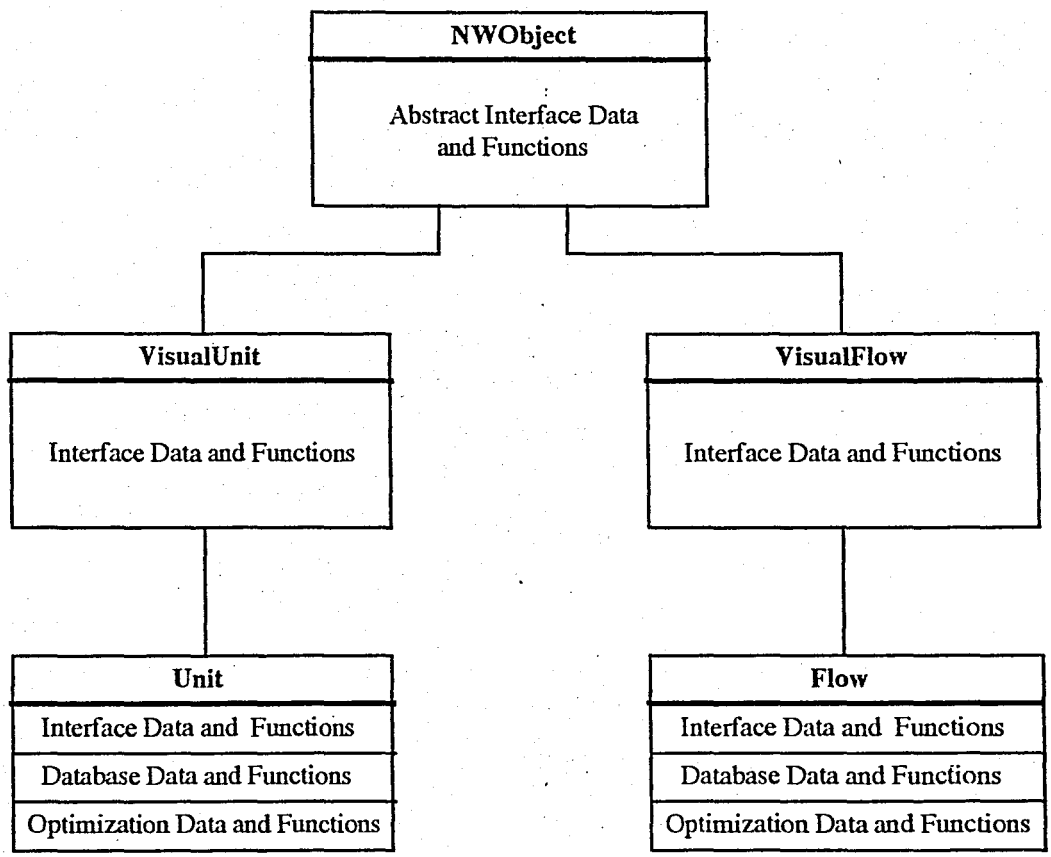


FIGURE 7.3.2 Inheritance of Properties of Unit and Flow Classes through their Parents

7.3.7.1. Interface Functions. Unit and Flow inherits its interface properties from their parents, and have no additional visual functions or visual data. However, these inherited visual properties are only associated with the graphic unit icons and flow arrows. The dialog

communication of flow and unit objects are defined separately in dialog classes which define the ways how the data exchange is performed.

A general outline of unit and flow dialog objects is given in the section reserved for dialog classes. A dialog class can be seen as a counterpart of a unit class or flow class. For instance, the FCC class and its dialog counterpart knows each other, hence the dialog class can be considered as a part of the FCC class, not a separate class from the viewpoint of object oriented programming. Whenever a dialog communication is requested, an FCC object creates its dialog counterpart, feed it with the necessary information, and give the control of the dialog management to it. The dialog object has the full responsibility for data input control and verification. When the communication ends, the dialog objects send the input to the FCC object, and destroys itself. The same process is valid for all unit and flow classes.

As explained in the section reserved for the user interface of AREMOS, the user is given feedbacks in case of invalid construction attempts. The validity checking is performed by the unit objects, and the associated rules are explained in sub-section 7.3.7.5. The main restriction is that there is a general sequence of process units through which crude oils are refined into final products. Therefore some product flow connections are invalid. For instance, an outflowing product from an FCC unit can not be feed into a distillation unit. In case of such an invalid connection, the associated unit objects communicate an error message to the interface, saying that the flow connection attempt is invalid. Another type of validity checking is performed by the unit and flow dialog objects which verify the input during the dialog communication. For instance, maximum capacity can not be less than minimum capacity for a unit, therefore a dialog data entry which results in such an invalidity is not allowed, and the user is given an associated error message. Finally, unit objects have an optimization function which check the validity of the data for the generation of the linear programming model. If unit objects detect errors, they communicate these error messages to the interface, and a list of errors are presented to the user. For instance, if the inflow to an FCC unit, which has a unit name of FCC-4, is deleted before the 'Compile' command is given, the FCC unit object communicates an error message saying 'Unit FCC-4 has outflows but no inflow'.

7.3.7.2. Database Functions. The database functions of the Flow class, except the ones already defined in VisualFlow, are simpler in nature. They only control the data and communicate with the head and tail units. Most of the Unit's database functions are designed to maintain the input and output flow objects. For instance when a flow is created by the user, the Network object notifies the head and tail units about this addition, and these units perform the flow addition operations. There are several database functions of the Unit class for such operations.

File storage and retrieval functions of unit and flow classes are designed using a well-known object oriented method. NWObject defines pure virtual 'save' and 'retrieve' functions. Each child class redefines these functions in such a way that the save function of the VisualUnit first calls the NWObject's save function and then save its own data, similarly Unit's save function first calls VisualFlow's save function, then save its own data, and this goes on through the child classes. This process is exactly the same for VisualFlow and its children. In this way, a new child doesn't need to save the data of its parent, just saves its own data and let the parent save its data. This process facilitates the generation of the save and retrieve functions of the child classes.

7.3.7.3. Optimization Functions. The data of the Flow class for optimization include a field for the variable name associated with the flow, and an index field that helps the generation of the variable name. The optimization functions of a flow are called only by the optimization functions of the head and tail units. The unit objects have the essential control of mathematical code generation, and the data and functions of a flow assist its head and tail units in this process. Index and variable name fields of a flow are defined by its tail unit during the compilation process, and used to be able to recall the produced variable names and indices.

Writing the mathematical programming code of a refinery model is the responsibility of unit objects. A determining property of the mathematical models of refinery processes is that each process unit generates its own code by only using the data of its inflows and outflows. It does not need to interact with other unit objects for this process. This very nature makes refinery modeling very appropriate for the object oriented approach, because otherwise a unit object would interact with other unit objects, and it would need to know the internal details of these units in many cases. Such kind of interaction is against to general principles of object oriented design, and do not exist in the design of AREMOS as will be discussed later in this section.

The optimization functions of the Unit class are defined mostly as purely virtual, which need redefinition in the child classes. These basically comprise a variable generation function, an objective function derivation function, a constraint derivation function and a validation function. Each child unit class, such as the FCC class, defines these functions according to the mathematical modeling of the specific process, and the specific data structure. These functions are explained in detail in the section reserved for the formulation of a linear programming model by unit objects.

7.3.7.4. Communication between Network Functions and Unit and Flow Functions. In the section reserved for the Network class, the categories of Network functions are explained, but how these functions communicate with the functions of the flow and unit objects are not

explained explicitly. As a requirement of the object oriented design, a network knows only the Unit and the Flow classes, therefore uses only the methods provided in these parent classes. The network sends generic messages like 'store yourself', 'display yourself', 'write your mathematical programming code' to a unit (or to a flow) without knowing its real class, and let it perform the rest. Each unit and flow knows how to respond to such a generic message, but the responses may be different. For instance, an FCC object and an HPDist object responds differently to the 'write your mathematical programming code' message.

The communication between the methods of the Network object and the methods of the unit and flow objects are depicted in Figure 7.3.3. As a general principle, each group of network functions interacts with the same group of flow or unit functions. The functions of a network have a generic nature, and the functions of units and flows determine the real implementation of a request from the network.

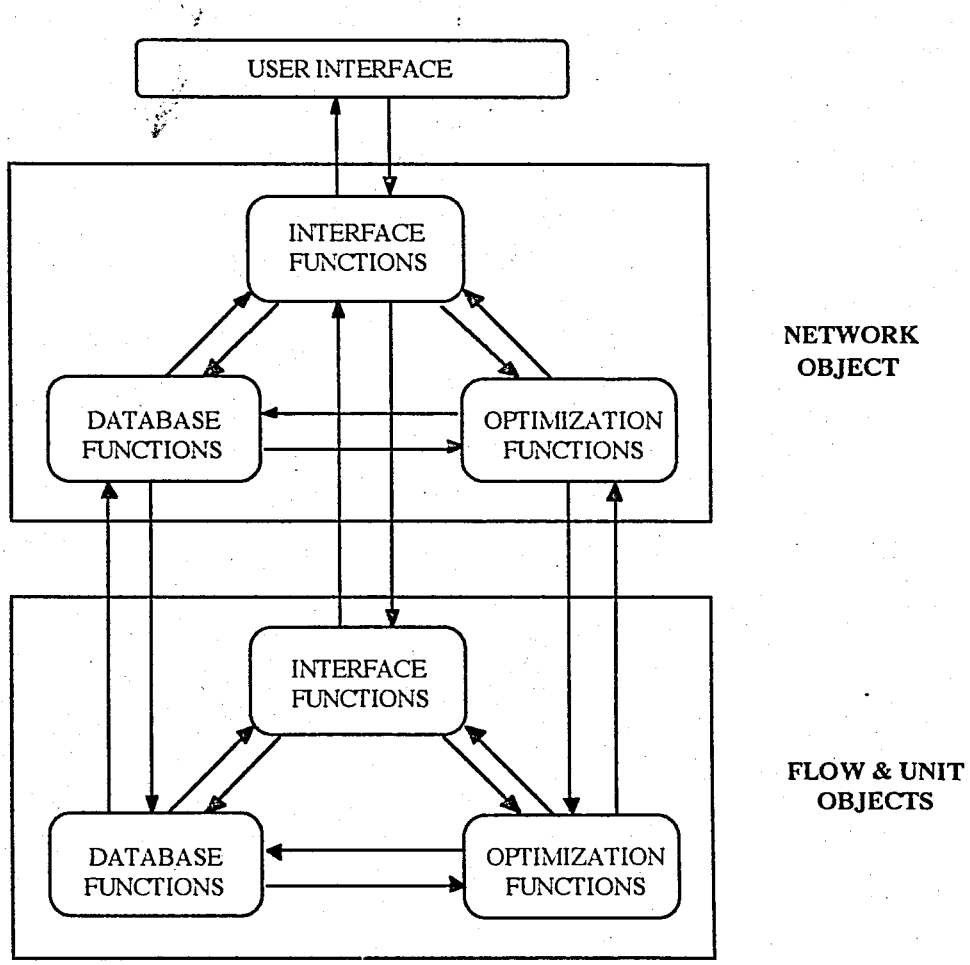


FIGURE 7.3.3 Communication between the Methods of Network Object and the Methods of Unit and Flow Objects

As previously explained, an important design consideration about the Network class is that only the Unit and the Flow classes are known to it. The same consideration is also valid for units and flows. A flow can not identify a specific class of unit or flow, ie. whether it is an HPDist object or FCC object, but identify it merely as a unit or flow. The same is also true for a unit, with the exception that HPDist, VacuumDist and CrudeTank objects can identify a CrudeFlow object as well. On the other hand, units and flows know the Network object they are kept in, which is obviously the only way they can answer the messages of the Network. On the whole, Network, Unit and Flow are the only classes among their children, which are known by the system.

7.3.7.5. Non-object Oriented Features. The design consideration explained in the previous paragraph has merit with respect to the object oriented approach, but this is not obtained without a price: Abstract design features had to be added to eliminate or minimize the needs of related objects to know about each other's class. This is required for cases where objects are closely related, such as refinery units that can not be arbitrarily connected by flows.

There is a general sequence of the refinery unit types from the entrance of the crude oils to the final products. For instance, a crude oil is first processed in an HP distillation unit than some products are sent to tanks and chemical processing units, and some to a Vacuum distillation unit. Therefore some connections turn out to be invalid. In order to accomplish the addition of this feature to the system, hypothetical refinery process levels are produced. These levels go from 0 to 7 successively, and unit can access the level of another unit without knowing its real class. By comparing the levels of the head and tail units of a flow, the validity of the flow connection can be checked. The general rule is that a flow can be sent from a unit of level x to a unit of level x or of a higher level. CrudeTank, HPDist, VacuumDist and Tank classes have levels 0, 1, 2 and 3 respectively. All the children of ChemProcUnit has level 4, Blender and GasoBlender has level 5, and FinalTank has level 7. ImportTank is given the level 0 like CrudeTank, and RedirectionPoint is given the level 6. In fact, RedirectionPoint is not a processing unit, and it has no requirement about the units it is connected with, hence its level number has no significance. However, a redirection point compares the levels of the units it connects, thus decides on the validity of the flows. The level number 6 is utilized only if redirection points are connected with each other. This situation requires a recursive level checking of the units. It is expected that possible new unit types can be levelled according to this general levelling rule. The HCC unit (Hydro-Cracking Unit), which is a new type of refinery unit that is planned to start running in 1996, has already given a level of 4, and added to the system design successfully.

A unit checks the validity of a new inflow connection by checking the level of the candidate tail unit, and validity of an outflow connection is determined similarly. In case of

an invalid connection the unit sends an error message to the network, which informs the user.

The NWObjectList of a Network object is made use of by its unit and flow objects for a better allocation of the visual resources, and for performing the tasks which needs reaching the instances of a specific class.

The instances of each class of unit or flow use the same resources, for instance, all flow objects use the same type of pen, brush and font. However each instance should normally create its own visual resources which means loss of memory and speed. Instead of creating the resources this way, they are allocated only once by the representative objects in the NWObjectList while initializing a Network object, and instances created afterwards use the resources of these objects. On the other hand, two different types of instances arise as a result of this design: The ordinary objects and the ones kept in the NWObjectList. Nevertheless, such a differentiation did not bring about a programming difficulty, and implemented by defining extra data and functions to be used only by the members of the NWObjectList.

The second usage of the NWObjectList is the localization of the non-object oriented tasks which are related with the instances of a specific class. The only such task encountered so far is the allocation of unique 3-digit variable indices to the instances of each unit class. For example, the instances of HPDist class in the system should be allocated different indexes like 1, 2, 3, etc. This task can not be performed by the Network object in an object oriented way. Instead, the representative HPDist object manages the allocation of these indices. In this way, the Network only sends generic messages to the NWObjectList, for allocating new indices for the created objects, or informs the list if a unit is deleted for an update of related indices. Turning back to the previous example, when a new HPDist object is created, the Network object finds the HPDist object in the NWObjectList, and let it dispatch an index number to the newly created HPDist object. Note that this operation does not require that the Network should distinguish the HPDist object.

7.3.8. CrudeFlow Class

CrudeFlow is generated as a result of a visual requirement. There can be a great deal of crude oils to be included into a refinery process model, that give rise to lots of crude flow drawings on the screen. This is a burden for the user, further it makes the screen full of crude flows, hence much more complicated. Instead of treating crude oil flows as ordinary flows, it is preferred to represent all crude flows to an HPDist unit as one arrow. Since crude oils flow only from the crude oil tank to the HPDist units, there are as many crude

connections as the HPDist units, therefore the visual model becomes quite understandable and manageable.

This visual preference is implemented by generating the CrudeFlow class as a child of the Flow class. The CrudeFlow inherits the visual characteristics of its parent, and has the crude connection arrow as its visual representation. However, a CrudeFlow object holds multiple flow objects, which are maintained as internal data members. CrudeFlow dialog box provides the means to manipulate the flows in a CrudeFlow object.

An advantage of the CrudeFlow class is that it is possible to impose mathematical constraints among the amounts of crude oils flowing to an HP distillation unit. These relations can be easily modeled within a CrudeFlow object without damaging the object oriented nature of the program. Such features are not included yet, except that the user can cancel some of the crude oil flows to an HP unit through the CrudeFlow dialog box.

7.3.9. Child Unit Classes

Children of the Unit class do not define any new visual function, but define their icon shapes, pens, brushes and fonts. These describe the visual appearance of a unit on the interface. New data and functions are added to model the specific process of each refinery unit, and pure optimization functions inherited form the Unit class are redefined according to the mathematical modeling of the process. The following is a summary of the designs of the child unit classes, and the next section gives a detailed view of the optimization functions.

The Unit class has only one abstract child class which is ChemProcUnit. ChemProcUnit serves as a base for chemical processing units whose common characteristic is the usage of input-output yield tables. In these units different types of chemical processes are applied on the inputs to obtain lighter products. The amounts of the output products can be defined as a percentage of the input in either volume or mass, and the mathematical model of the process is derived using this relationship. As a restriction, a chemical processing unit accepts only one product flow, but have many product yields.

The CrudeTank and FinalTank classes do not represent real refinery units. Only one instance of the CrudeTank and the FinalTank are allowed in a refinery network. The CrudeTank object models the crude oil feedstock of a refinery, and the FinalTank object represents a hypothetical outlet point to which all the final products are connected. The CrudeTank and the FinalTank objects provide the means of modeling the data about crude oils and final products. Most important data are obviously crude oil costs and final product sales prices. Besides, minimum and maximum consumption figures for crude oils, and minimum and maximum market demands for the final products can be specified. The default

values for these minimums and maximums are zero and infinity (which is defined as a very big number). By redefining these values, lower and upper bounds can be imposed on each final product and each crude oil.

The ImportTank is similar to the CrudeTank and used to represent the feedstock of import products, however each ImportTank object models the feedstock of only one type of import product. Upper and lower bound values can be specified for the import product consumption like crude oils.

The Tank class represents real refinery tanks for semi-finished products. On the other hand, tank objects produce no mathematical constraint or objective function figure, instead behave as pipes with unlimited flux capacity. Addition of an upper flux capacity is a possible extension to the Tank class, however this requires special care because the upper bound to be added should correctly represent the long-run restriction of a tank imposed on the refinery process.

The HPDist and the VacuumDist classes model two different types of distillation units. Crude oils are first fed to the HP distillation units in which the crude oils are refined into different products. Then the residual of this process is sent to Vacuum distillation units for further refinement. The yield ratios of the refinement process in HP and Vacuum units heavily depend on the specific properties of crude oils. Therefore both of these classes know the CrudeFlow class, and demand properties of crude oils during modeling. Instances of these classes make use of complex yield tables for modeling the input-output yield relations.

The Blender and the GasoBlender classes model the blending and gasoline blending operations. These operations are mostly performed as the last refinery operations to obtain the final products with required quality limitations. A blending unit takes the quality specifications of input product flows, ie. density, viscosity index, etc., and the quality requirements of its output flows (as upper and lower bounds). It uses these values to generate the mathematical model of the input-output relationships in the form of mathematical constraints. Gasoline blending is a special type of blending operation in which extra operations such as tetraethyllead addition are performed. As a result, the GasoBlender class defined as a child of the Blend, so that it inherits all general blending data and methods.

Finally, RedirectionPoint derives from the Unit class, however it is not a real unit. It represents a flow connection in the refinery which may have multiple inflows and multiple outflows. It is mainly used to redirect a flow to different units, or to combine similar product flows in one flow stream.

7.4. Formulation of a Refinery Linear Programming Model by Unit Objects

The previous section was a general summary of the child unit classes. It is clear that all the data of these classes are defined to specify the refinery process that are performed in these units. The optimization functions of a unit use these data for the mathematical formulation of the process. All the unit classes redefine the optimization functions that are defined in the parent Unit class, and add supplementary ones to these basic functions.

Unit optimization functions basically comprise a validation function, a variable generation function, an objective function derivation function, and a constraint derivation function. These functions of unit objects in a network model are managed by the optimization management functions of the associated network object. All unit objects validate their flow connections and their data before the generation of the linear programming model. Once the model is validated, all unit objects generate their variable names, and store them in a list name 'VarExp'. As explained in the sub-section reserved for the optimization management functions of the Network class, this list is formed for processing the optimal solutions produced by the optimization module. The objective function derivation functions and constraint derivation functions are called after the generation of the variable names. Each unit object is responsible for generating its mathematical constraints and objective function part. Unit objects make use of well-established refinery linear programming techniques in their objective function and constraint derivation functions. These techniques were successfully applied to TÜPRAŞ İzmit Refinery in a previous case study [43], and the validation of the generated linear programming model provided strong insights that the results obtained through the model were consistent with the real production figures of the refinery.

The following sub-sections explain variable generation, validation, objective function generation and constraint generation functions of refinery unit objects. While explaining the objective function and constraint generation of unit objects, the examples will be given from the sample TÜPRAŞ İzmit Refinery model developed in AREMOS, which is depicted in Figures 6.1.1 and 6.1.2. Appendix C gives the linear programming model file of this refinery network model. The objective function and the constraints of this linear programming model are produced by the unit objects of the refinery network model. Besides, Appendix D gives the meanings of the variables used in the linear program generation and the names of the unit objects which produce the variables. Finally, Appendix E gives the associated optimum results report.

7.4.1. Validity Checking

The validation function of a unit object is responsible for checking the validity of inflow and outflow connections, and the validity of the data with respect to the linear program generation. If an invalidity is detected, an associated error message is produced to the user for the correction of the error.

Refinery units should have valid product flow connections in order to be able to produce their constraints and objective function terms. The validity requirements related with the flow connections are as follows:

(a) A crude oils tank should have at least one outflowing crude oil.
(b) An import tank should have an outflowing import product.
(c) Final products tank should have at least one inflowing final product inflow.
(d) Other units should have both inflows and outflows, or none. If a unit has no inflows and no outflows, it is not connected with the production network, therefore it is ignored in the linear program generation process.

The other type of requirements are related with the data part of unit objects, and are as follows:

(a) The units with utility consumption figures should have a positive total utility cost per input charge.
(b) Blending and gasoline blending units should have valid upper and lower quality specification limits. For example, if product A with a density of 0.9 and product B with a density of 0.8 are blended, the density of the resulting blend can vary between 0.8 and 0.9. Therefore an upper density limit which is less than 0.8, and a lower density limit which is greater than 0.9 are infeasible. The same requirement applies to the other quality types.

7.4.2. Variable Generation

The crucial operation for optimization is the generation of variables. Each unit object is responsible for creating its own variables. In order for each unit to create distinct variable names, a unique letter is assigned to each class, and a unique 3-digit index is assigned to each unit object during the run time. By means of the letter and the index, a unit object is

able to produce its variable names without bothering about the variable names that other units produce. The variables of a unit refer to the variables corresponding to the unit's outflows and the variables which are generated internally by the unit (ie. in a blending unit, amount of inflow i blended in outflow j is expressed as a variable which is produced internally by the blending unit).

A restriction encountered in the generation of variable names is that the solver limits the variable names to eight characters. In fact, this is a restriction of the MPS format for linear programming models. Jones argues that this is an important difficulty in the generation of variable names for large problems [33]. Such a difficulty is encountered in gasoline blending units in the linear programming formulation of a refinery model by AREMOS. The following paragraphs explain how this difficulty is overcome, as well as how other units generate their variable names.

In refinery units except for the gasoline blending unit and the crude oils tank, the first four letters of a variable name define the specific unit that is producing the variable name, and following two letters are reserved for defining the outflows. Outflows are enumerated starting from 0 to represent them in the last two letter space. For example F00403 is the variable associated with the third outflow of an FCC unit with the distinct index 004. It is clear that 6 letters are enough for generating such a variable name.

The crude oils tank object is an exception to this rule, since there is only one crude oils tank in a refinery network as explained in the child unit classes section. Therefore, the crude oils tank object makes use of only its unique letter 'C ' without any need to its 3-digit number. The crude oils tank enumerates its crude oils and crude oil flows for variable generation. For example, C001 represents the total amount of crude oil 1 used in the refinery production, and C001001 represents the amount of crude oil 1 flowing in the first crude oil stream, which is modeled as a CrudeFlow object. A CrudeFlow object defines the crude flow connection between the crude oils tank and an HP distillation unit. It models multiple crude oil flows to an HP Unit. The CrudeFlow class section can be referred for detailed information on crude oil flows.

In blending units additional variables with 8 characters are generated for formulating the blending process. B0030102, for instance, is produced by a blending unit with the index 003, and expresses the amount of first inflow blended in the third outflow of the unit. A gasoline blending unit makes use of the variable generation techniques of a blending unit. In fact, the GasoBlender class is the child of the Blender class. However, it still needs extra variables with an additional index in order to formulate tetraethyllead (TEL) addition as linear programming constraints. As a solution to this problem, distinct 3-digit index of variable names is reduced to a 1-digit index in gasoline blending units. Since TEL formulation is a rather complicated process, the format of the new variables will not be explained here in detail. The result of this reduction is that two letter fields are saved and these are used for

defining the additional index fields for the new variables. Consequently, total number of gasoline blending units are restricted to ten in a refinery model, while this bound is a thousand for other units. Nevertheless, in a refinery normally one gasoline blending unit is needed and therefore, ten can be accepted as a sufficiently large number. Likewise, a thousand is a sufficiently large bound for the total number of other units.

7.4.3. Objective Function Derivation

The entire objective function of the linear programming model of a refinery is produced by requesting each unit object to write its objective function terms in a text file. The objective of the model is to maximize the daily net profit of the refinery and can be expressed as follows:

MAXIMIZE

Net Profit = Sales Revenues - Crude Oil Costs - Import Product Costs
- Unit Utility Consumption Costs (7.4.1)

In this formulation,

(a) Sales revenues are produced by final products tank object;

(b) Crude oil costs are produced by crude oils tank object;

(c) Import product costs are produced by import tank objects;

(d) Unit utility consumption costs are produced by the unit objects which have utility consumptions. These are all chemical processing unit objects, HP distillation unit objects, and Vacuum distillation unit objects.

(a) Sales revenues: The objective function part produced by the final products tank involves the sales revenues from the final products and can be formulated as follows:

$$\text{Sales Revenues} = \sum_i (SPFP)_i \times (FP)_i \quad (7.4.2)$$

where,

$(FP)_i$ is the variable representing the volume amount of final product i, and
$(SPFP)_i$ is the selling price of final product i.

Example: The objective function part produced by the final products tank of the TÜPRAŞ İzmit refinery model is as follows:

+52.51 B04801+134.25 R03401+70.27 B04802+114.56 B04803
+149 R03001+86.32 R04001+119.17 R03902+132.05 R03801
+27.73 R04201+225.23 B02101+156.8 B04901+164.31 B04902 (7.4.3)
+166 B04903+244.55 G05001+281.32 G05002+242 G05003
+20 L01603

In this formulation, variables represent final products flowing into the final products tank (variable meanings, and the tail units of these product flows can be found in Appendix D), and coefficients are the prices per volume of final products.

(b) Crude oil costs: The objective function part produced by the crude oils tank involves the costs of the crude oils and can be formulated as follows:

$$\text{Crude Oil Costs} = \sum_i (COC)_i \times (CO)_i \qquad (7.4.4)$$

where,
$(CO)_i$ is the variable representing the volume amount of crude oil i, and
$(COC)_i$ is the cost of crude oil i.

Example: The objective function part produced by the crude oils tank of the TÜPRAŞ İzmit refinery model includes three terms corresponding to three crude oils used in the modeling:

-92.970001C002-88.589996C001-90.040001C003 (7.4.5)

Here, variables C001, C002 and C003 represent the volume amounts of crude oil types Sarir, Essider, and Iran Light, respectively. The associated coefficients are the costs per volume of crude oils.

(c) Import product costs: The cost terms associated with the import products are generated by import tanks. However, since one import tank object corresponds to one import product type, each import tank object produces just one objective function term. Summation of these terms for all import tanks forms the total import product costs:

$$\text{Import Product Costs} = \sum_i (\text{CIM})_i \times (\text{IM})_i \qquad (7.4.6)$$

Here,
$(\text{IM})_i$ is the variable representing the volume amount of import product i, and
$(\text{CIM})_i$ is the cost of import product i.

Example: There is no import tank used in the modeling of TÜPRAŞ İzmit Refinery. To give an example, suppose that the management considers purchasing naphtha to be used in gasoline blending operation. In order to model this consideration, a naphtha import unit is generated and its outflow is connected to the gasoline blending unit. Then, a purchase (import) cost is specified for the naphtha import. Suppose that the purchase cost per volume of naphtha import is 40. Then the associated objective function term generated by the import tank is as follows:

$$-40\ \text{I00101} \qquad (7.4.7)$$

In this formulation, I00101 represents the volume amount of naphtha import which is drawn from the import tank and used in the gasoline blending operation.

(d) Unit utility consumption costs: Unit utility consumption costs are the costs of running the refinery units. Utility consumptions of a unit depend on the weight amount of the input charge to the unit. Unit utilities include electricity, burner oil, cooling water, and water steams of different pressures, and many catalysts. Crude oils tank, final products tank, refinery tanks for semi-finished products (instance of the Tank class), blending units

and redirection points have no utility consumptions. The total utility consumption cost of other unit types can be written as follows:

$$\text{Unit Utility Consumption Costs} = \sum_i \sum_j (TUC)_i \times d_{ij} \times (IC)_{ij} \qquad (7.4.8)$$

Here,
$(IC)_{ij}$ is the variable representing the volume amount of the inflow charge j to refinery unit i, d_{ij} is the density of the inflow charge j to refinery unit i, and
$(TUC)_i$ is the total utility cost of refinery unit i per input volume charge. $(TUC)_i$ is a monetary amount calculated by refinery unit i, using the utility consumption figures of the unit and utility cost figures of the associated refinery network model.

Example: The HP-2 distillation unit object of the TÜPRAŞ İzmit refinery model is selected for explaining how a unit object generates its unit utility consumption terms for the objective function. The HP-2 unit object first calculates the $(TUC)_{HP\text{-}2}$ term. The consumption figures of HP-2 unit object are specified in the unit utility dialog box, and kept in the UnitUtils object of the unit. The appearance of the unit utility dialog box of HP-2 object is depicted in Figure 6.1.7. Utility consumption costs, on the other hand, are specified in the utility cost data dialog box of the refinery network, and kept in the NWUtils object of the network object. The utility consumptions for the HP-2 unit and the related cost figures are as follows:

TABLE 7.4.1 Utility Cost and Utility Consumption Figures for an HP Distillation Unit

Utility	Consumption/M.Ton charge	Utility Cost
Burner Oil	0.0190 metric ton	50.8400 $/metric ton
Electricity	4.8250 Kwh	0.0400 $/Kwh
Cooling Water	2.0110 metric ton	0.3500 $/metric ton
Steam 550	0.0800 metric ton	6.8800 $/metric ton
Steam 150	0.0030 metric ton	6.6900 $/metric ton
Steam 50	0.0139 metric ton	6.4600 $/metric ton
Inhibitor	0.0056 kg	2.3530 $/metric ton
NaOH	0.0152 kg	0.3920 $/metric ton
NH3	0.0040 kg	0.3920 $/metric ton
Demulsifier	0.0050 kg	1.4710 $/metric ton

In this table, consumption units, such as metric ton and kg, and the monetary cost unit are given for clarity of the calculations. On the other hand, these are not necessary in other formulations, and not given explicitly. It is clear from the table that $(TUC)_{HP\text{-}2}$ is calculated by multiplying the two figures in each line, and summing up the multiplication results over all lines:

$$(TUC)_{HP\text{-}2} = 0.0190 \times 50.840 + 4.8250 \times 0.040 + \ \ldots \ + 0.0050 \times 1.471 \qquad (7.4.9)$$

$$(TUC)_{HP\text{-}2} = 2.5511$$

There are 3 crude oil flows into the HP-2 unit, these are namely Sarir, Essider, and Iran Light crude oils. The variables representing these crude oil flows are C001002, C002002, and C003002, and the densities of these crude oils are 0.838, 0.8358, and 0.8555, respectively. Therefore the objective function part is formulated as follows:

$$\begin{aligned}(\text{Utility Cost})_{HP\text{-}2} = & - (TUC)_{HP\text{-}2} \times 0.838 \times C001002 \\ & - (TUC)_{HP\text{-}2} \times 0.8358 \times C002002 \\ & - (TUC)_{HP\text{-}2} \times 0.8355 \times C003002 \end{aligned} \qquad (7.4.10)$$

$$\begin{aligned} = & - 2.1378\ C001002 \\ & -2.1322\ \ C002002 \\ & -2.1825\ \ C003002 \end{aligned} \qquad (7.4.11)$$

7.4.4. Constraint Derivation

The entire constraint part of a refinery linear programming model is produced by requesting each unit object to write its constraints in a text file. Constraints of a unit express the process description, the quality descriptions of the unit's outflows and some other restrictions.

Process description constraints describe the whole product flow scheme in a refinery process network. These constraints are material balance equations, yield equations, and unit capacity constraints. Quality description constraints are used in blending operations, and

other restrictions apply to crude oils, import products and final products. The constraint types used by different unit objects are as follows:

(a) Material balance equations are produced by the tank and redirection point objects;
(b) Yield equations are produced by chemical processing unit objects, HP distillation unit objects, and Vacuum distillation unit objects;
(c) Unit capacity constraints are produced by chemical processing unit objects, HP distillation unit objects, and Vacuum distillation unit objects;
(d) Quality description constraints are produced by blending and gasoline blending unit objects;
(e) Other restrictions are produced by final products tank, import product tank, and crude oils tank objects.

(a) Material balance equations are the simplest process description constraints. These are used by the tank and redirection point objects and have the following form:

$$\sum_i (\text{Outflow})_i - \sum_j (\text{Inflow})_j = 0 \qquad (7.4.12)$$

In this equation,
$(\text{Outflow})_i$ is the variable representing the volume amount of inflow i to the unit, and
$(\text{Inflow})_j$ is the variable representing the volume amount of outflow j from the unit.

Example: In the TÜPRAŞ İzmit Refinery model, the tank unit object which models the kero tank in the refinery produces the following material balance constraint:

$$\text{T01101-R01302} = 0 \qquad (7.4.13)$$

Here, the variables T00110 represents the volume amount of the outflowing product, and R01302 represents the volume amount of the inflowing product.

(b) Yield equations express the mathematical relationships between the amounts of inflows and amounts of outflows of a refinery unit. There are two major types of yield equations: The ones used by the chemical processing units and the ones used by the

distillation units. The yield equations used by the chemical processing units have the following general form:

$$(\text{Outflow})_i - (\text{OFP})_i \times \text{Inflow} = 0 \qquad (7.4.14)$$

In this equation,
$(\text{Outflow})_i$ is the variable representing the volume amount of outflow (yield) i of the unit,
Inflow is the variable representing the volume amount of inflow to the unit, and
$(\text{OFP})_i$ is the volume yield ratio of inflowing product for outflow i.

Example: The 3-D desulphurizer unit object of the TÜPRAŞ İzmit Refinery model, which has one inflow and two outflows, produces the following equations:

$$100\ D02601 - 3.35\ R02702 = 0 \qquad (7.4.15)$$
$$100\ D02602 - 96.65\ R02702 = 0 \qquad (7.4.16)$$

Here, the variables D02601 and D02602 represent the volume amounts of two outflowing products from the unit, and R02702 represents the volume amount of the inflowing product to the unit. These equations state that 96.65 per cent of the inflow yields one product, and 3.35 percent yield the other product, as a result of the chemical process in the unit.

The yield equations for the distillation units have a different form, because the output yield ratios of these units depend on the properties of the crude oils. The yield equations used by the distillation units have the following form:

$$\sum_i (\text{OFP})_{ij} \times (\text{CO})_i - (\text{OF})_j = 0 \qquad (7.4.17)$$

where,
$(\text{CO})_i$ is the variable representing the volume amount of crude oil i fed into the distillation unit,
$(\text{OF})_j$ is the variable representing the volume amount of outflow (yield) j of the unit, and
$(\text{OFP})_{ij}$ is the volume yield ratio of crude oil i for outflow j.

Example: The HP-2 distillation unit object of the TÜPRAŞ İzmit Refinery model has 3 inflowing crude oils, and 7 outflowing products (The dialog box of this distillation unit is depicted in Figure 6.1.5). The associated yield equations are as follows:

$$100\ H00201 - 1.3\ C002002 - 1.3\ C001002 - 1.4\ C003002 = 0 \quad (7.4.18)$$

$$100\ H00202 - 52.5\ C002002 - 56.7\ C001002 - 54.63\ C003002 = 0 \quad (7.4.19)$$

$$100\ H00203 - 0.2\ C002002 - 0.07\ C003002 = 0 \quad (7.4.20)$$

$$100\ H00204 - 11.2\ C002002 - 11.2\ C001002 - 11.1\ C003002 = 0 \quad (7.4.21)$$

$$100\ H00205 - 6.8\ C002002 - 6.8\ C001002 - 8.8\ C003002 = 0 \quad (7.4.22)$$

$$100\ H00206 - 12.5\ C002002 - 12.5\ C001002 - 12.5\ C003002 = 0 \quad (7.4.23)$$

$$100\ H00207 - 12\ C002002 - 11.5\ C001002 - 11.5\ C003002 = 0 \quad (7.4.24)$$

In Equation 7.4.18, H00201 represents the volume amount of one outflow from the HP-2 unit, and C001002, C002002 and C003002 represent the volume amounts of three different crude oils, namely Sarir, Essider, and Iran Light crude oils, fed into the unit. The coefficients are the percent volume yield ratios of the crude oils. For instance, the yield ratio of 1.3 at the beginning of C003002 means that 1.3 volume percent of the Essider crude oil fed into the HP-2 unit is transformed into the product represented by H00201. The other equations are written for the remaining six outflows of the HP-2 distillation unit. Note that the yield coefficients are related with both the crude oil types and the technical conditions of the specific distillation unit. Therefore these coefficients are kept as the data of the distillation unit objects, and directly entered in the associated distillation unit dialog boxes.

(c) Unit capacity constraints apply to distillation units and chemical processing units. These constraints impose lower and upper bounds to the daily production of units as follows:

$$\sum_i (\text{Inflow})_i \leq \text{MaxCap}$$

$$\sum_i (\text{Inflow})_i \geq \text{MinCap} \quad (7.4.25)$$

where,

$(\text{Inflow})_i$ is the variable representing the volume amount of inflow i to the unit, and

MaxCap and MinCap are the maximum and minimum bounds for the daily production.

Example: The HP-2 distillation unit given in the previous example has 6000 and 8000 as its lower and upper bounds for the daily production. The associated inequalities have the following form:

$$+C003002+C001002+C002002 >= 6000$$
$$+C003002+C001002+C002002 <= 8000 \qquad (7.4.26)$$

Here, left parts represent the total volume input charge to the distillation unit, and right parts are the minimum and maximum bounds for the daily production.

(d) The quality description constraints are used in blending and gasoline blending units. Certain quality specifications are set for some products and these specifications are satisfied by blending similar types of products with different quality levels. The quality types included in AREMOS are density, viscosity, sulphur content and octane number. Blending and gasoline blending unit objects produce the following complementary equations before writing the quality description constraints:

$$(\text{Inflow})_i - \sum_i (\text{InOut})_{ij} = 0$$
$$(\text{Outflow})_j - \sum_j (\text{InOut})_{ij} = 0 \qquad (7.4.27)$$

In these equations,
$(\text{Inflow})_i$ is the variable representing the volume amount of inflow i,
$(\text{Outflow})_j$ is the variable representing the volume amount of outflow j, and
$(\text{InOut})_{ij}$ is the variable representing the volume amount of inflow i blended into outflow j.

Example: F.Oil blending unit is a blending unit of TÜPRAŞ İzmit Refinery Model. It has two inflowing products to be blended, and three blends as outflows. Let's rename these inflows as inflow-1, inflow-2, and the outflows as outflow-1, outflow-2 and outflow-3, for better understanding the following equations produced by the F.Oil blending unit:

$$D02602 - B0490101 - B0490102 - B0490103 = 0 \quad (7.4.28)$$

$$D03102 - B0490201 - B0490202 - B0490203 = 0 \quad (7.4.29)$$

$$B04901 - B0490101 - B0490201 = 0 \quad (7.4.30)$$

$$B04902 - B0490102 - B0490202 = 0 \quad (7.4.31)$$

$$B04903 - B0490103 - B0490203 = 0 \quad (7.4.32)$$

In Equation 7.4.28, D02602 is the volume amount of inflow-1, and B0490101, B0490102 and B0490103 represent the volume amounts of inflow-1 blended into outflow-1, outflow-2 and outflow-3, respectively. Equation 7.4.29 is written for inflow-2, and is very similar to the first one. In Equation 7.4.30, B04901 represents the volume amount of outflow-1, and B0490101 and B0490201 represent the volume amounts of inflow-1 blended into outflow-1 and inflow-2 blended into outflow-1, respectively. Equations 7.4.31 and 7.4.32 are very similar to the third one, and written for outflow-2 and outflow-3.

In a blending operation the minimum and maximum density levels for an outflow are specified in terms of the blended inflows as follows:

$$\begin{aligned} (Mind)_j \times (Outflow)_j - \sum_i d_i \times (InOut)_{ij} &\leq 0 \\ (Maxd)_j \times (Outflow)_j - \sum_i d_i \times (InOut)_{ij} &\geq 0 \end{aligned} \quad (7.4.33)$$

where,
$(Outflow)_j$ is the variable representing the volume amount of outflow j,
$(Mind)_j$ is the minimum density level for outflow j,
$(Maxd)_j$ is the maximum density level for outflow j,
$(InOut)_{ij}$ is the variable representing the volume amount of inflow i blended into outflow j, and
d_i is the density of inflow i.

Similarly, the minimum and maximum viscosity index levels for an outflow are specified as follows:

$$\begin{aligned} (Minv)_j \times (Outflow)_j - \sum_i v_i \times (InOut)_{ij} &\leq 0 \\ (Maxv)_j \times (Outflow)_j - \sum_i v_i \times (InOut)_{ij} &\geq 0 \end{aligned} \quad (7.4.34)$$

where,
$(Outflow)_j$ is the variable representing the volume amount of outflow j,
$(Minv)_j$ is the minimum viscosity index level for outflow j,
$(Maxv)_j$ is the maximum viscosity index level for outflow j,
$(InOut)_{ij}$ is the variable representing the volume amount of inflow i blended into outflow j, and
v_i is the viscosity index of inflow i.

Note that viscosity indices used in the refineries are specifically produced for refinery usage such that they blend linearly by volume.

The quality type of sulphur content does not blend by volume but by mass. Consequently, additional density terms appear in the blending constraints that specify the minimum and maximum sulphur content levels for an outflow:

$$(Mins)_j \times d_j \times (Outflow)_j - \sum_i s_i \times d_i \times (InOut)_{ij} \leq 0$$
$$(Maxs)_j \times d_j \times (Outflow)_j - \sum_i s_i \times d_i \times (InOut)_{ij} \geq 0 \qquad (7.4.35)$$

where,
$(Outflow)_j$ is the variable representing the volume amount of outflow j,
$(Mins)_j$ is the minimum sulphur content level for outflow j,
$(Maxs)_j$ is the maximum sulphur content level for outflow j,
$(InOut)_{ij}$ is the variable representing the volume amount of inflow i blended into outflow j,
d_i is the density of inflow i, and
d_j is the density of outflow j.

Example: All the above quality description constraints have a very similar form. Therefore, density level constraints produced by the F.Oil blending unit of the TÜPRAŞ İzmit Refinery model are selected as the demonstrative examples for quality description constraints:

$$0.785\ B04902 - 0.78299\ B0490102 - 0.79\ B0490202 <= 0 \qquad (7.4.36)$$
$$0.82\ B04902 - 0.78299\ B0490102 - 0.79\ B0490202 >= 0 \qquad (7.4.37)$$

In these inequalities, B04902 represents the volume amount of outflow-2, and B0490102 and B0490202 represent the volume amounts of inflow-1 blended into outflow-2 and

inflow-2 blended into outflow-2, respectively (as explained in the previous example). In Inequality 7.4.36, 0.785 is the minimum density level for outflow-2, and 0.78299 and 0.79 are the densities of the inflow-1 and inflow-2. Similarly, 0.82 is the maximum density level for outflow-2 in Inequality 7.4.37.

Blending operation to satisfy required octane levels of gasoline products, such as premium and normal gasolines, is only performed in the gasoline blending unit. It has a different nature, since a catalyst, namely tetraethyllead, is incorporated in the blending process. Addition of tetraethyllead increases the octane number, but it does not linearly affect the octane level of the gasoline blend. Therefore a special treatment is required for linearizing the mathematical formulation of the gasoline blending operation. The technique used in for this purpose is developed by Kawaratani and et al. [5]. Since this technique is quite complicated, it will not be given explicitly here. Instead the Kawaratani's paper can be referred for a detailed discussion.

(e) Other restrictions that apply to refinery production are the limitations on the amount of crude oil input, import products and final products. Therefore, the associated constraints are produced by the crude oils tank, the final product tank and the import tanks of a refinery model. These constraints have a simple form as follows:

$$(\text{Product Flow})_i \geq \text{MinLimit}$$
$$(\text{Product Flow})_i \leq \text{MaxLimit} \qquad (7.4.38)$$

In these inequalities;
$(\text{Product Flow})_i$ is the variable representing the volume amount of crude oil i, import product i, or final product i,
MinLimit and MaxLimit are the minimum and maximum daily consumption limits of crude oil i or import product i, or the minimum and maximum daily demand figures for final product i.

Example: In the linear programming model of TÜPRAŞ İzmit refinery in Appendix C, the last 18 constraints are produced by the final products tank, and impose lower and upper bounds to the daily production amounts of the final products. These bounds can also be regarded as minimum and maximum demands for the final products.

7.5. Maintenance of Objects at Run Time

Having covered the object class hierarchy and the detailed designs of the classes used in AREMOS, an overall picture of how the objects are maintained at run time is needed in order to have an enhanced view of the system.

As previously emphasized, the network, the unit and the flow objects are the central objects of the system. Moreover the RefApp objects, and the RefMDIFrame and the RefMDIClient objects can be regarded as the important objects of the system with respect to their functions, however most of these functions are inherited from their parents, which are standard ObjectWindows classes, and work internally without any need of the programmer's modification.

Figure 7.5.1 illustrates how the objects of the system are maintained at run time. The basic characteristic is the usage of different lists to keep the networks, units and flows. The RefApp object sets up the connection of AREMOS with the Windows operating system. It only maintains the RefMDIFrame object. The RefMDIFrame object maintains an internal list of the Network objects, and a RefMDIClient object whose primary role is the behind-the-scenes management of the Network objects. The RefMDIFrame together with the RefMDIClient manage the Network list.

Elements of the list maintained by the RefMDIFrame are composed of the Network objects who appear with their modeling windows (or as an icon that represents the minimized state of a window) on the screen. When the user clicks on a refinery modeling window, the control passes to the associated Network object, and from that point on any user message is sent to this object by the RefMDIClient until another refinery window is clicked and hence takes the control. When a refinery object is closed by the user, the associated object is detached from the list and destroyed. For this reason, a network model should be saved before closing if it is to be retrieved for further study. Similarly a network scheme is either to be developed from scratch or retrieved from a file.

The primary elements of a Network are flows and units. A flow object defines a connection between two different unit objects, and normally all of the unit objects are connected in this way. The resulting network builds up a refinery process scheme. For the overall management, a Network object keeps all its units in one list and all its flows in another list. Any operation on units and flows are performed through traversing these lists.

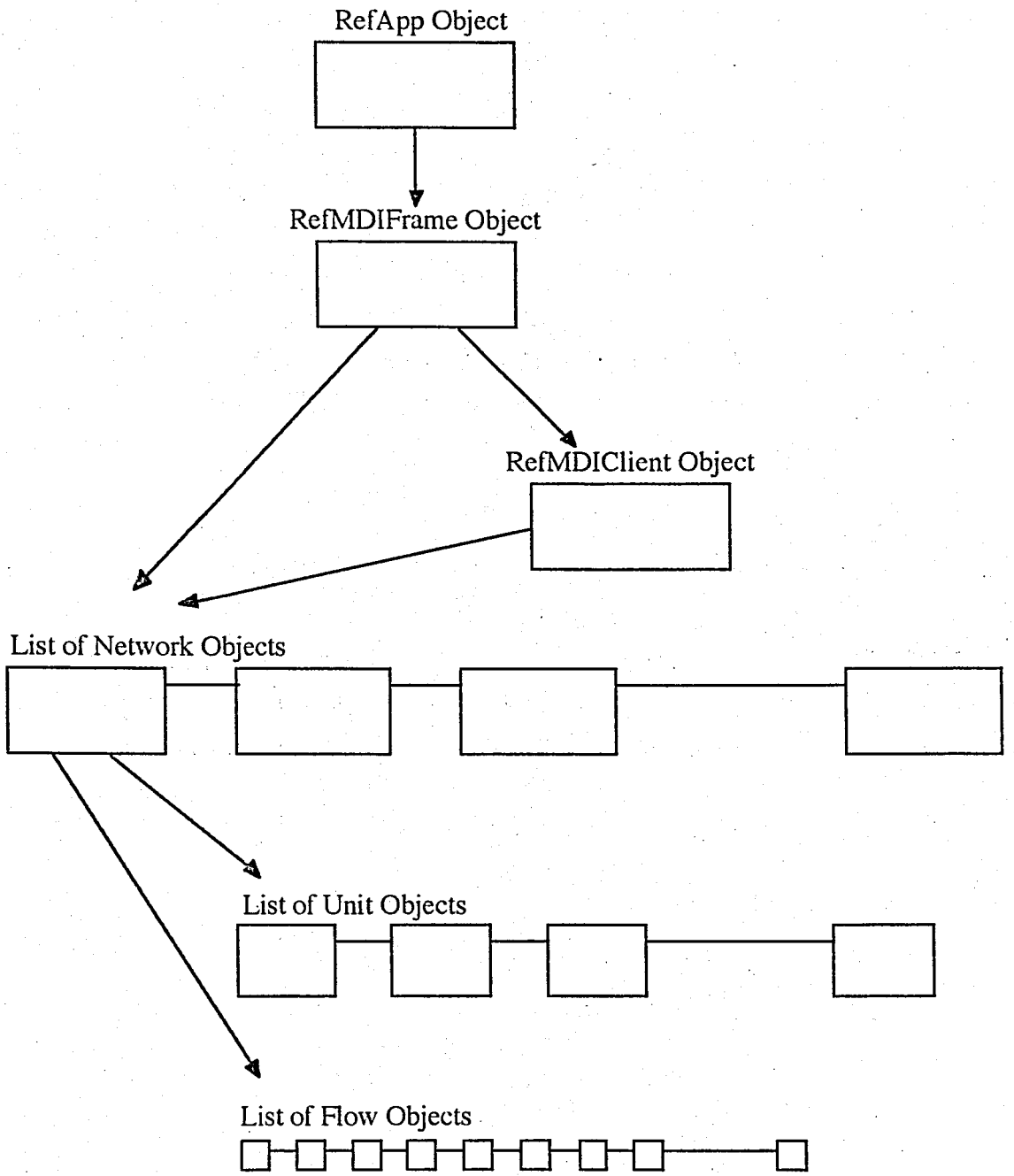


FIGURE 7.5.1 Maintenance of Objects at Run Time

A Network object doesn't maintain the connections among units and flows directly. In fact, these are not known to the Network at all. Instead, each unit knows its flow connections and each flow knows its unit connections. Consequently a unit itself manages its connections with the flows and similarly a flow manages the connections it makes with the head and tail units. According to the user manipulations, the Network object sends

generic messages like 'accept this flow as your inflow', 'accept this unit as your head unit', 'delete yourself', and the rest is the responsibility of the units and the flows.

A flow knows its head and tail units through two pointers as depicted in Figure 7.5.2, and these are the only connections of a flow with other objects. On the other hand, a unit can have more than one inflow and outflow, and maintains two lists for keeping them. As a result, a flow object is hold by the inflow list of its head unit and the outflow list of its tail unit. Figure 7.5.3 demonstrates this relationship.

Since a flow object is also a member of the global flow list of the network, three different lists maintain a flow object at the same time. Therefore special care is needed for the management of flow objects. For instance, when the user creates a new product flow, the network sends 'accept this flow as your inflow' and 'accept this flow as your outflow' messages to the candidated head and tail units. If they validate the connection, the network adds the flow object in its flow list, otherwise deletes the created flow. The addition of the flow into the units' internal flow lists is performed by the units. The deletion of a flow is a little more complicated. When the user deletes a flow, the Network object first detaches the associated flow object from its flow list, and sends a 'delete yourself' message to the flow object. Then the flow object first sends 'detach me from your flow list' message to its head and tail units. Only after these detachments the flow object destroys itself.

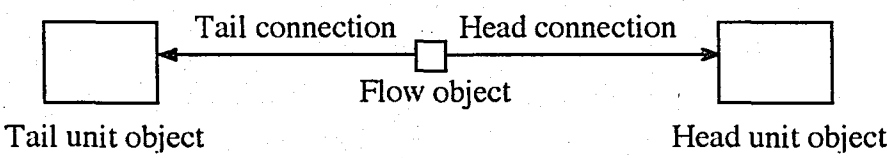


FIGURE 7.5.2 Unit Connections of a Flow Object

A unit object interacts with its inflows and outflows through traversing its flow lists while a flow has a direct connection to its head and tail units, and acts as a bridge between them. For instance, the shared flow object in Figure 7.5.3 defines a product flow from one unit object to the other, and these unit objects interact with each other by means of this flow object.

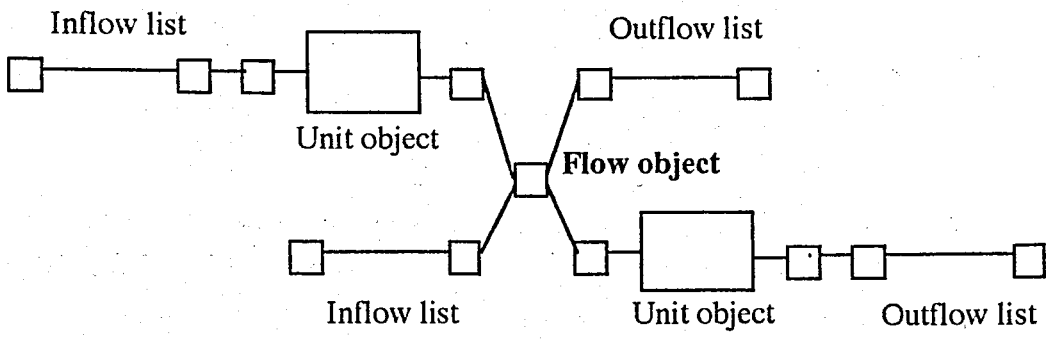


FIGURE 7.5.3 Maintenance of a Flow Object by its Tail and Head Units

The dialog objects have almost as equal importance as the unit and flow objects have, however they temporarily exist during the dialog input-output operations. When a dialog process is initiated (by pushing a button or double-clicking screen objects) a dialog object is created, and when the process is over it is deleted. For this reason, they are not shown in Figure 7.5.1. Dialog objects are called directly by the screen objects for which a dialog request is made by the user. They can be seen as the interface counterparts of their 'underlying' unit and flow objects.

Another important object that is not shown in Figure 7.5.1 is the NWObjectList of A Network. The NWObjectList can be regarded as a supplementary object whose primary role is the localization of non-object oriented features that exist in the system design of AREMOS. It doesn't hold the real data pertaining to the refinery models, however helps allocating visual resources to unit and flow objects. The NWObjectList is set up when a Network object is initialized, and kept without any addition or deletion during the life cycle of the Network.

8. CONCLUSIONS

The intention in the development of AREMOS is to provide the management of a refinery with an effective decision support tool for obtaining the optimal production policy. It brings ideas together from visual interactive modeling methodology and object oriented design and programming, and uses well-established refinery mathematical programming techniques as the underlying optimization paradigm. AREMOS is generated based on the results of a case study in TÜPRAŞ İzmit Petroleum Refinery, but it is designed in such a way to cover the general characteristics of a typical petroleum refinery, hence it is also applicable to other petroleum refineries.

Unlike other decision support tools covered in the literature survey section, AREMOS puts emphasis on the design of the user interface. The interface part represents a conceptual model frequently used to describe the refinery process: A network model which shows units (or processes) as nodes and product flows as arcs, thus has an object-based representation. The interface part enables the user to model the process as a visual network as neatly as possible. This general representation constitutes the base for the interface design of AREMOS. Dialog boxes, menus, and other visual utilities come as complementary visual aids to this network modeling process. The significant characteristics of the visual interface are that the user can represent the whole process as a picture on the screen, and easily access any part of it. Furthermore, it can play with several different refinery models simultaneously. This helps the user to maintain an overall conceptual picture of the process in mind.

An important interface design consideration in AREMOS is that it should separate and hide the requirements specific to mathematical programming from the user as much as possible. This is fully achieved for the refinery model development phase; general refinery engineering knowledge is sufficient for developing a refinery production scheme, but a minimum mathematical programming knowledge is still needed to interpret the optimal results.

In order to test the effectiveness of the interface features of AREMOS, different models of TÜPRAŞ İzmit Refinery have been built and tried in laboratory environment. Having built similar refinery models in LINDO and GAMS, it is our experience that modeling in AREMOS makes it a lot easier to understand the relations among different processes, to detect modeling errors and to compare different scenarios. However, AREMOS needs to be tested in the real refinery environment to observe its real advantages and disadvantages. We believe no disadvantage in connection with the general design characteristics will appear, however missing parts are certain to exist, since AREMOS is a

prototype system and many facilities are to be added in order to make it work with full performance.

Besides the benefits of the user interface, the object oriented system design of AREMOS provides substantial advantages on the programmer side. The design of the system is built in a way that is not oriented to the optimization techniques utilized, instead the real process units and real product flows are modeled as the objects which are the building stones of the system. This makes the system indifferent to the underlying operations research technique used. Therefore the general structure of the system forms a suitable platform for the addition of other techniques such as simulation, non-linear programming, or mixed-integer programming.

Applicability of these paradigms in an object oriented fashion is a different topic of concern. This is accomplished for the case of mathematical programming by making use of a special nature of the refinery mathematical modeling process that the refinery unit objects don't need to interact with each other for mathematical code generation. In fact, mathematical programming can be argued as the most difficult modeling paradigm for the application of object oriented methodology, as it does not use the concept of objects at all. Other techniques can be applied with less effort, and clever ways can be generated to handle the non-object oriented tasks that might arise.

More important than the applicability of different techniques is the facilitated maintenance of the system as a result of its object oriented design. The maintenance process requires at least as much effort as the system development process requires. The complexity considerably increases in large systems like AREMOS, consequently the maintenance might turn out to be a very time consuming task. However, the object oriented nature of AREMOS provides effective means to modify a part of the system or to add new facilities (or new type of units, etc.) without any need to deal with unnecessary parts of the system code. The object oriented design makes AREMOS easily extendable and changeable without damaging the robustness of the system. These features will be very beneficial for improving the system according to the specific requirements of the refinery, and there are already lots of prospective extensions as will be discussed in the future work section.

The object oriented nature brings another important advantage in the interface design. Because the flow and the unit objects have their visual counterparts on the screen, the management of the interface is performed in a very natural way. Simply, responsibility of the screen objects are given to their underlying objects, and the refinery object maintains the screen by means of its units and flows. It sends the user requests as generic messages to the underlying flows and units, and lets them perform the detailed interface operation. As a result, the responsibility of interface management is shared in a balanced way. This correspondence between the underlying objects and the visual objects enable the programmer easily maintain the interface part of the objects.

9. FUTURE WORK

AREMOS is designed using Visual Interactive Modeling methodology which requires that the design should start, continue and end focusing on the needs of the user. Therefore development of visual interactive systems require intensive cooperation of the programmer with the end user. The development of AREMOS has started with the needs of the management of TÜPRAŞ İzmit Refinery. At present it reached a level that the resulting software can be regarded as the first prototype to be proposed to the refinery management. Yet there will be successive prototypes on the way to produce a real decision support system for in-house usage, and the development of these prototypes is very likely to necessitate a closer collaboration with the management. Consequently, AREMOS should first be tested by the refinery management, and their comments and requests should be reflected as improvements to the current system. The system developer should be aware of the fact that what seems to be a good improvement may not be favored by the management. This picture summarises the aspect of the future work on AREMOS regarding the end user.

There are prospective improvements which can be proposed as the future steps of the AREMOS project. It is a good idea to enhance the optimization facilities of the system as the first step. Two types of enhancement are possible: For the existing underlying mathematical model, the ways that the data are retrieved and presented can be improved according to the comments of the refinery management. For instance, the dialog box designs and the optimal results presentation process can be improved. Addition of sensitivity analysis facilities should also be considered in this context. Besides, new constraints and objective function items can be added to the underlying mathematical model, and existing ones can be improved accordingly. Inclusion of the constraints that restrict the proportion of a crude oil (or a combination of crude oils) in the crude feed to a distillation unit can be given as examples for these type of additions. Note that such constraint and objective function enhancements bring about updates in the interface as a consequence.

As a next step, mathematical programming formulations can be enhanced by the addition of nonlinearities, but this will require a nonlinear programming solver, and the solution time will considerably increase as a result.

The addition of new operations research techniques to the existing system can be considered as important steps toward a comprehensive decision support system. Simulation seems to be the most promising one among all, since simulation techniques have been used in petroleum refineries extensively as well as mathematical programming techniques for a long time. More importantly, object oriented visual interactive simulation packages, most of which make use of visual network-based representations, have already been proved to be

successful, and gained a wide acceptance in the marketplace. These preliminaries are already enough to argue that a simulation module can be successfully added to AREMOS. In a petroleum refinery the production policy is determined for a planning horizon (ie. a 3 month period) by means of optimization. On the other hand, the refinery production conditions have a very dynamic and unpredictable nature. For instance, crude oil prices or product sales prices are likely to change considerably within a planning horizon, or unexpected breakdowns may occur in the process. A simulation module can be a great help in doing the necessary adjustments in the production plan in such circumstances. In fact, this is the common practice in petroleum refineries. Further, simulation is a different operations research paradigm from optimization, and the parallel usage of these two techniques is likely to enable the refinery management to obtain more optimal and reliable production plans. Because, each technique provides different means to discover different aspects of the refinery production, combination of results from the two techniques are reasonably expected to provide better production policies. The simulation module will appear as a new management component of the object oriented management system in Figure 6.2.1.

A different type of extension to AREMOS is incorporation of an expert system, that is the addition of intelligence. Integration of decision support systems and expert systems is a fairly new and promising research area. AREMOS primarily needs two types of expertise. Refinery-specific expertise is needed in the model development phase in order to have a better model in the sense that it represents the real refinery process as closely as possible. Refinery expertise is especially helpful in eliminating the conception errors and fulfilling the expert adjustments in modeling. On the other hand, mathematical programming expertise is needed to aid the user in interpreting the optimal results, discovering the sources of infeasibilities, and discovering the modeling errors that can not be captured by simple methods. It is also needed to perform comprehensive sensitivity analyses on the optimum production results.

APPENDIX A.

BORLAND C++ COMPILER, OBJECTWINDOWS CLASS LIBRARY AND CONTAINER CLASS LIBRARY

Borland C++ & Application Frameworks 3.1 is a comprehensive C and C++ program development environment which supports generating programs for Microsoft Windows version 3.1. It comes with supplementary programs; a Windows oriented integrated program development environment, a debugger which supports Windows programming, a resource workshop which is used to generate visual resources, such as dialog boxes, bitmaps, icons and cursors, to be incorporated into developed C++ codes are among these supplementary utilities.

Borland C++ & Application Frameworks 3.1 provides two built-in class libraries. ObjectWindows class library provides the programmer with an object oriented program development platform under Windows. ObjectWindows is a complete collection of classes that describe standard Windows features. By using the property of inheritance, the programmer can take advantage of pre-written code that performs the repetitive work required to write Windows applications, and can design the interface part of the programs in a completely object oriented way. ObjectWindows classes ease the program development under Windows by providing a consistent, intuitive, and simplified interface to Windows. They supply the behavior for window management and message processing, and structuring a Windows application, which are performed quite differently from the way a standard C++ program works. Please refer to ObjectWindows User's Guide [41] for further information on the usage of these classes.

The other library is the Container class library which provides several well-known data structures in the form of object classes. These include Array, SortedArray, List, DoubleList, Btree (Binary tree), Stack, Queue, Stack, HashTable, and some others as well as different iterator classes to traverse instance of these objects. Each class is equipped with the related functions performed on the included elements. The only requirement for the elements to be hold by these classes is that they should derive from the Object which serves as the base for all the Container classes. A container class can hold instances of different Object-derived classes, that is it can have a heterogenous collection of elements. There is no limitation for the number of elements to be maintained, except for the total available memory. Borland C++ Programmer's Guide [46] reserves an extensive section for the Container class library , which can be referred for further information.

APPENDIX B.

AREMOS INSTALLATION GUIDE

AREMOS comes with a number files that can be found in the attached floppy diskette. These are :

1. aremos.exe : The AREMOS executable Windows Application file.
2. lindo.exe : The LINDO mathematical program solver.
3. bwcc.dll : The dynamic link library necessary for running aremos.exe under Windows.
4. loadbwcc.exe : The executable file which loads bwcc.dll into the Windows operating system.
5. linrun.bat : The batch file for running LINDO.
6. linbatch.bat : The LINDO batch file for taking, solving and reporting the results of the refinery mathematical model file generated by aremos.exe.
7. linrun.pif : Program information file that introduces linrun.bat to Windows.
8. linbatch.pif : Program information file that introduces linbatch.pif to Windows.
9. *.hlp files : Help files to be used by aremos.exe.
10. *.net files : Refinery network model files for demonstration.

Requirements:

1. An IBM compatible computer with 80-386 processor plus math-coprocessor or 80-486 processor or a higher version (Intel Pentium series).
2. Microsoft Windows Operating System with version 3.1 or higher.
3. At least 2 MB RAM.

Installation Steps:

1. Copy BWCC.DLL into Windows SYSTEM subdirectory.
   copy a:\bwcc.dll c:\windows\system
2. Copy LOADBWCC.EXE into Windows subdirectory.
   copy a:\loadbwcc.exe c:\windows
3. Copy AREMOS.EXE and HELP files, *.hlp, and LINDO BATCH files, *.bat, into a subdirectory.

copy a:\aremos.exe c:\aremos1
copy a:*.hlp c:\aremos1
copy a:*.bat c:\aremos1

4. Add LOADBWCC.EXE to LOAD statement of WIN.INI.
   load=loadbwcc.exe
5. Copy pif files into Windows subdirectory.
   copy a:*.pif c:\windows
6. Copy lindo.exe to any directory and make the necessary updates in lindo.pif file.
   ie. copy a:\lindo.exe c:\aremos
7. Restart Windows. Normally it will work under 386-enhanced mode. Note that AREMOS will not work under protected or standard mode.
8. Create a new group using File | New menu commands.

Additions :

1. The linear programming model file of the last compiled refinery network model can be found in a file named 'lpmodel.out'. The information about the variables appearing in this file is stored in another file named 'variable.out'.

2. Once the refinery model is constructed and optimization is performed, the report of optimization results are stored in files with extensions ".rep". For instance report file for mymodel1.net will be stored in mymodel1.rep, hence can be examined using any text editor. However there is also an online report utility within the modeling system for this purpose. Using a text editor on the other hand may help taking printout of the report or play with the format of the report.

APPENDIX C.

A REFINERY LINEAR PROGRAMMING MODEL PREPARED BY AREMOS

MAX

-92.970001C002-88.589996C001-90.040001C003
-2.1322C002002-2.1378C001002-2.1825C003002
-4.7005C002001-4.7129C001001-4.8113C003001
-5.0689H00603
-2.33H00202
-10.6345R03002
-5.3848R02402
-20.154R03003
-2.5231U03701
-2.3505U02901
-1.9144R02702
-2.7387R02002
-40.9615R02703
-10.1933R03402
-12.8753R03403
-7.0618R04701
-0.7392G10101T1-1.4784G10101T2-1.6489G10101T3-2.2744G10101T4
-0.7392G10201T1-1.4784G10201T2-1.6489G10201T3-2.2744G10201T4
-0.7392G10301T1-1.4784G10301T2-1.6489G10301T3-2.2744G10301T4
-0.7392G10401T1-1.4784G10401T2-1.6489G10401T3-2.2744G10401T4
-0.7392G10501T1-1.4784G10501T2-1.6489G10501T3-2.2744G10501T4
-0.7392G10102T1-1.4784G10102T2-1.6489G10102T3-2.2744G10102T4
-0.7392G10202T1-1.4784G10202T2-1.6489G10202T3-2.2744G10202T4
-0.7392G10302T1-1.4784G10302T2-1.6489G10302T3-2.2744G10302T4
-0.7392G10402T1-1.4784G10402T2-1.6489G10402T3-2.2744G10402T4
-0.7392G10502T1-1.4784G10502T2-1.6489G10502T3-2.2744G10502T4
+52.51B04801+134.25R03401+70.27B04802+114.56B04803
+149R03001+86.32R04001+119.17R03902+132.05R03801
+27.73R04201+225.23B02101+156.8B04901+164.31B04902

+166B04903+244.55G05001+281.32G05002+242G05003
+20L01603

SUBJECT TO

C002-C002001-C002002=0
C001-C001001-C001002=0
C003-C003001-C003002=0
+C002002+C001002+C003002>=6000
+C002002+C001002+C003002<=8000
100H00201-1.3C002002-1.3C001002-1.4C003002=0
100H00202-52.5C002002-56.7C001002-54.63C003002=0
100H00203-0.2C002002-0.07C003002=0
100H00204-11.2C002002-11.2C001002-11.1C003002=0
100H00205-6.8C002002-6.8C001002-8.8C003002=0
100H00206-12.5C002002-12.5C001002-12.5C003002=0
100H00207-12C002002-11.5C001002-11.5C003002=0
+C002001+C001001+C003001>=10500
+C002001+C001001+C003001<=15000
100H00601-1.7C002001-1C001001-1.1C003001=0
100H00602-0.2C002001-0.07C003001=0
100H00603-55.1C002001-59.3C001001-57.23C003001=0
100H00604-11.7C002001-11.7C001001-11.6C003001=0
100H00605-8.5C002001-6.5C001001-8.5C003001=0
100H00606-11.5C002001-11C001001-11C003001=0
100H00607-11C002001-10.5C001001-10.5C003001=0
+H00603<=7000
100V00701-23.1C002001-27.3C001001-28C003001=0
100V00702-14.5C002001-14.5C001001-13C003001=0
100V00703-17.5C002001-17.5C001001-16C003001=0
+H00202<=4000
100V00501-28.3C002002-28.3C001002-30.43C003002=0
100V00502-19.8C002002-18.5C001002-14.7C003002=0
100V00503-9.9C002002-9.9C001002-9.5C003002=0
T01801-V00502-V00702=0
T00901-H00204-H00604=0
T01001-H00205-H00605=0
T01101-R01301=0

+R03002<=2600
100U03701-100R03002=0
+R02402<=1400
100U02301-99.1R02402=0
+R03003<=2000
100U02901-99.1R03003=0
+U03701<=2600
100P03601-10.5U03701=0
100P03602-3U03701=0
100P03603-86.5U03701=0
+U02901<=1900
100P03501-12.57U02901=0
100P03502-8.43U02901=0
100P03503-79U02901=0
+R02702<=1200
100D02601-3.35R02702=0
100D02602-96.65R02702=0
+R02002<=3600
100D02801-4.74R02002=0
100D02802-95.26R02002=0
+R02703<=2800
100D03101-3R02703=0
100D03102-96.65R02703=0
+R03402<=1900
100F03201-19.15R03402=0
100F03202-2.55R03402=0
100F03203-12.43R03402=0
100F03204-20.34R03402=0
100F03205-40.28R03402=0
+R03403<=2700
100F03301-8.2R03403=0
100F03302-4.6R03403=0
100F03303-23.5R03403=0
100F03304-14.57R03403=0
100F03305-44.16R03403=0
+R04701<=1500
100L01601-97R04701=0
100L01602-1R04701=0

100L01603-2R04701=0
F03301-B0480101-B0480102-B0480103=0
F03201-B0480201-B0480202-B0480203=0
V00701-B0480301-B0480302-B0480303=0
V00501-B0480401-B0480402-B0480403=0
B04801-B0480101-B0480201-B0480301-B0480401=0
B04802-B0480102-B0480202-B0480302-B0480402=0
B04803-B0480103-B0480203-B0480303-B0480403=0
0.95B04803-1.05B0480103-0.9746B0480203-0.9418B0480303-0.9524B0480403>=0
R02001-B0210101=0
R02701-B0210201=0
D02802-B0210301=0
F03204-B0210401=0
F03304-B0210501=0
B02101-B0210101-B0210201-B0210301-B0210401-B0210501=0
D02602-B0490101-B0490102-B0490103=0
D03102-B0490201-B0490202-B0490203=0
B04901-B0490101-B0490201=0
B04902-B0490102-B0490202=0
B04903-B0490103-B0490203=0
0.75B04901-0.78299B0490101-0.79B0490201<=0
0.82B04901-0.78299B0490101-0.79B0490201>=0
0.785B04902-0.78299B0490102-0.79B0490202<=0
0.82B04902-0.78299B0490102-0.79B0490202>=0
P03503-G0500101-G0500102-G0500103=0
P03603-G0500201-G0500202-G0500203=0
F03205-G0500301-G0500302-G0500303=0
F03305-G0500401-G0500402-G0500403=0
R02501-G0500501-G0500502-G0500503=0
G05001-G0500101-G0500201-G0500301-G0500401-G0500501=0
G05002-G0500102-G0500202-G0500302-G0500402-G0500502=0
G05003-G0500103-G0500203-G0500303-G0500403-G0500503=0
0.71G05001-0.7846G0500101-0.7978G0500201-0.7144G0500301-0.7325G0500401
-0.6829G0500501<=0
0.75G05001-0.7846G0500101-0.7978G0500201-0.7144G0500301-0.7325G0500401
-0.6829G0500501>=0
0.725G05002-0.7846G0500102-0.7978G0500202-0.7144G0500302-0.7325G0500402
-0.6829G0500502<=0

0.76G05002-0.7846G0500102-0.7978G0500202-0.7144G0500302-0.7325G0500402
-0.6829G0500502>=0
G0500101-G10101T0-G10101T1-G10101T2-G10101T3-G10101T4=0
G0500201-G10201T0-G10201T1-G10201T2-G10201T3-G10201T4=0
G0500301-G10301T0-G10301T1-G10301T2-G10301T3-G10301T4=0
G0500401-G10401T0-G10401T1-G10401T2-G10401T3-G10401T4=0
G0500501-G10501T0-G10501T1-G10501T2-G10501T3-G10501T4=0
G0500102-G10102T0-G10102T1-G10102T2-G10102T3-G10102T4=0
G0500202-G10202T0-G10202T1-G10202T2-G10202T3-G10202T4=0
G0500302-G10302T0-G10302T1-G10302T2-G10302T3-G10302T4=0
G0500402-G10402T0-G10402T1-G10402T2-G10402T3-G10402T4=0
G0500502-G10502T0-G10502T1-G10502T2-G10502T3-G10502T4=0
91G05001-96G10101T0-96.563G10101T1-96.821G10101T2-96.862G10101T3
-97G10101T4-98G10201T0-98.563G10201T1-98.821G10201T2-98.862G10201T3
-99G10201T4-93G10301T0-93.563G10301T1-93.821G10301T2-93.862G10301T3
-94G10301T4-93.5G10401T0-94.063G10401T1-94.321G10401T2-94.362G10401T3
-94.5G10401T4-68G10501T0-68.563G10501T1-68.821G10501T2-68.862G10501T3
-69G10501T4<=0
95G05002-96G10102T0-96.563G10102T1-96.821G10102T2-96.862G10102T3
-97G10102T4-98G10202T0-98.563G10202T1-98.821G10202T2-98.862G10202T3
-99G10202T4-93G10302T0-93.563G10302T1-93.821G10302T2-93.862G10302T3
-94G10302T4-93.5G10402T0-94.063G10402T1-94.321G10402T2-94.362G10402T3
-94.5G10402T4-68G10502T0-68.563G10502T1-68.821G10502T2-68.862G10502T3
-69G10502T4<=0
R04301-P03501-D02601-R04401-R04601=0
R04201-R04301-R00401=0
R04101-L01601-P03602=0
R04001-R04101-P03502=0
R02501-R02401=0
R02401+R02402+R02403-T00901=0
R01701-H00601-H00201=0
R00801-H00602=0
R00401-R00801-H00203=0
R01201+R01202-H00206=0
R01301+R01302-H00606-R01201=0
R01401-R01302-H00607-V00703-R01501=0
R01501-R01202-H00207-V00503=0
R02701+R02702+R02703-T01101=0

```
R03001+R03002+R03003-T01001=0
R03401+R03402+R03403-T01801=0
R03801-R02403-R03901=0
R03901+R03902-U02301=0
R04401-F03202-R04501=0
R04501-F03302-L01602=0
R04601-P03601-D03101-D02801=0
R04701-R01701-F03203-F03303=0
R02001+R02002-R01401=0
B04801>=100
R03401>=100
B04802>=100
B04803>=100
R03001>=100
R04001>=100
R03902>=100
R03801>=100
R04201>=100
B02101>=100
B04901>=100
B04902>=100
B04903>=100
G05001>=1000
G05002>=1000
G05003>=100
G05003<=200
L01603>=10
END
```

APPENDIX D.

THE VARIABLES GENERATED FOR A REFINERY MODEL BY AREMOS

Variable	Meaning of the Variable	Owning Unit Name
R02002	Dies-36D	P8
R02001	Die	P8
R04701	LPG	P25
R04601	FG	P24
R04501	FG	P23
R04401	FG	P22
R03902	SWLSR	P16
R03901	LSR	P16
R03801	LSR	P17
R03403	HVGO-7fcc	P15
R03402	HVGO-4fcc	P15
R03401	HVGO	P15
R03003	HSR-6U	P14
R03002	HSRto36U	P14
R03001	HSRFinal	P14
R02703	Kero-6D	P13
R02702	Kero-3D	P13
R02701	Kero-Mot	P13
R01501	Diesel	P6
R01401	Diesel	P5
R01302	Kero	P4
R01301	Kero	P4
R01202	Kero	P3
R01201	Kero	P3
R00401	FG	P1
R00801	FG	P2
R01701	LPG	P7
R02403	LSR	P12
R02402	LSR	P12
R02401	LSR	P12

R02501	Naphta	Naphta Unit
R04001	LPG	P18
R04101	LPG	P19
R04201	FG	P20
R04301	FG	P21
G0500503	Amount of Naphta used in Platformate	G. Blending Station
G0500403	Amount of WCN used in Platformate	G. Blending Station
G0500303	Amount of WCN used in Platformate	G. Blending Station
G0500203	Amount of Plat used in Platformate	G. Blending Station
G0500103	Amount of Plat used in Platformate	G. Blending Station
G0500502	Amount of Naphta used in Premium Gaso	G. Blending Station
G0500402	Amount of WCN used in Premium Gaso	G. Blending Station
G0500302	Amount of WCN used in Premium Gaso	G. Blending Station
G0500202	Amount of Plat used in Premium Gaso	G. Blending Station
G0500102	Amount of Plat used in Premium Gaso	G. Blending Station
G0500501	Amount of Naphta used in Normal Gaso	G. Blending Station
G0500401	Amount of WCN used in Normal Gaso	G. Blending Station
G0500301	Amount of WCN used in Normal Gaso	G. Blending Station
G0500201	Amount of Plat used in Normal Gaso	G. Blending Station
G0500101	Amount of Plat used in Normal Gaso	G. Blending Station
G05003	Platformate	G. Blending Station
G05002	Premium Gaso	G. Blending Station
G05001	Normal Gaso	G. Blending Station
B0490203	Amount of Kero used in JP 4	Kero Blending
B0490103	Amount of Kero used in JP 4	Kero Blending
B0490202	Amount of Kero used in JET A-1	Kero Blending
B0490102	Amount of Kero used in JET A-1	Kero Blending
B0490201	Amount of Kero used in Kerosene	Kero Blending
B0490101	Amount of Kero used in Kerosene	Kero Blending
B04903	JP 4	Kero Blending
B04902	JET A-1	Kero Blending
B04901	Kerosene	Kero Blending
B0210501	Amount of LCGO used in Motorin	Mot Blending
B0210401	Amount of LCGO used in Motorin	Mot Blending
B0210301	Amount of Diesel used in Motorin	Mot Blending
B0210201	Amount of Kero-Mot used in Motorin	Mot Blending
B0210101	Amount of Die used in Motorin	Mot Blending
B02101	Motorin	Mot Blending

B0480403	Amount of VRSD used in Ozel Kal Yak	F.Oil Blending
B0480303	Amount of VRSD used in Ozel Kal Yak	F.Oil Blending
B0480203	Amount of F.Oil used in Ozel Kal Yak	F.Oil Blending
B0480103	Amount of F.Oil used in Ozel Kal Yak	F.Oil Blending
B0480402	Amount of VRSD used in F.Oil 6	F.Oil Blending
B0480302	Amount of VRSD used in F.Oil 6	F.Oil Blending
B0480202	Amount of F.Oil used in F.Oil 6	F.Oil Blending
B0480102	Amount of F.Oil used in F.Oil 6	F.Oil Blending
B0480401	Amount of VRSD used in Burner F.Oil	F.Oil Blending
B0480301	Amount of VRSD used in Burner F.Oil	F.Oil Blending
B0480201	Amount of F.Oil used in Burner F.Oil	F.Oil Blending
B0480101	Amount of F.Oil used in Burner F.Oil	F.Oil Blending
B04803	Ozel Kal Yak	F.Oil Blending
B04802	F.Oil 6	F.Oil Blending
B04801	Burner F.Oil	F.Oil Blending
L01603	Sulphur	7 LPG
L01602	FG	7 LPG
L01601	LPG	7 LPG
F03305	WCN	7 FCC
F03304	LCGO	7 FCC
F03303	LPG	7 FCC
F03302	FG	7 FCC
F03301	F.Oil	7 FCC
F03205	WCN	4 FCC
F03204	LCGO	4 FCC
F03203	LPG	4 FCC
F03202	FG	4 FCC
F03201	F.Oil	4 FCC
D03102	Kero	6 D
D03101	FG	6 D
D02802	Diesel	36 D
D02801	FG	36 D
D02602	Kero	3 D
D02601	FG	3 D
P03503	Plat	6 P
P03502	LPG	6 P
P03501	FG	6 P
P03603	Plat	36 P

P03602	LPG	36 P
P03601	FG	36 P
U02901	LSR	6 U
U02301	LSR	3 U
U03701	HSR	36 U
T01101	Kero	Kero Tank
T01001	HSR	HSR Tank
T00901	LSR	LSR Tank
T01801	HVGO	HVGO Tank
V00503	LVGO	2 V
V00502	HVGO	2 V
V00501	VRSD	2 V
V00703	LVGO	5 V
V00702	HVGO	5 V
V00701	VRSD	5 V
H00607	Diesel	5 HP
H00606	Kero	5 HP
H00605	HSR	5 HP
H00604	LSR	5 HP
H00603	ATSR	5 HP
H00602	FG	5 HP
H00601	LPG	5 HP
H00207	Diesel	2 HP
H00206	Kero	2 HP
H00205	HSR	2 HP
H00204	LSR	2 HP
H00203	FG	2 HP
H00202	ATSR	2 HP
H00201	LPG	2 HP
C003	Crude Oil iran light	CRUDE OIL TANK
C001	Crude Oil sarir	CRUDE OIL TANK
C002	Crude Oil essider	CRUDE OIL TANK
C002002	Crude Oil essider to 2 HP	CRUDE OIL TANK
C001002	Crude Oil sarir to 2 HP	CRUDE OIL TANK
C003002	Crude Oil iran light to 2 HP	CRUDE OIL TANK
C002001	Crude Oil essider to 5 HP	CRUDE OIL TANK
C001001	Crude Oil sarir to 5 HP	CRUDE OIL TANK
C003001	Crude Oil iran light to 5 HP	CRUDE OIL TANK

APPENDIX E.

AN OPTIMAL RESULTS REPORT PREPARED BY AREMOS

Summary Report:
Current model has the following feasible solution

Objective funtion value: 1592626

P8 :	
Dies-36D	0
Die	7392.2241
P25 :	
LPG	868.5976
P24 :	
FG	157.2848
P23 :	
FG	70.8171
P22 :	
FG	119.2671
P16 :	
SWLSR	100
LSR	0
P17 :	
LSR	1726.531
P15 :	
HVGO-7fcc	1350.677
HVGO-4fcc	1900
HVGO	100

P14 :	
HSR-6U	0
HSRto36U	1497.95
HSRFinal	100
P13 :	
Kero-6D	0
Kero-3D	310.3983
Kero-Mot	0
P6 :	
Diesel	1668.571
P5 :	
Diesel	7392.2241
P4 :	
Kero	2102.9629
Kero	310.3983
P3 :	
Kero	0
Kero	952.381
P1 :	
FG	40.6465
P2 :	
FG	25.4083
P7 :	
LPG	315.0186
P12 :	
LSR	1726.531
LSR	100.9082

LSR	512.283
Naphta Unit :	
Naphta	512.283
P18 :	
LPG	887.4782
P19 :	
LPG	887.4782
P20 :	
FG	327.5966
P21 :	
FG	286.9502
G. Blending Station :	
Amount of Naphta used in Platformate	200
Amount of WCN used in Platformate	0
Amount of WCN used in Platformate	0
Amount of Plat used in Platformate	0
Amount of Plat used in Platformate	0
Amount of Naphta used in Premium Gaso	132.6714
Amount of WCN used in Premium Gaso	596.4588
Amount of WCN used in Premium Gaso	242.9897
Amount of Plat used in Premium Gaso	997.6687
Amount of Plat used in Premium Gaso	0
Amount of Naphta used in Normal Gaso	179.6116
Amount of WCN used in Normal Gaso	0
Amount of WCN used in Normal Gaso	522.3303
Amount of Plat used in Normal Gaso	298.0581
Amount of Plat used in Normal Gaso	0
Platformate	200
Premium Gaso	1969.7889
Normal Gaso	1000

Kero Blending :	
Amount of Kero used in JP 4	0
Amount of Kero used in JP 4	100
Amount of Kero used in JET A-1	0
Amount of Kero used in JET A-1	100
Amount of Kero used in Kerosene	0
Amount of Kero used in Kerosene	100
JP 4	100
JET A-1	100
Kerosene	100
Mot Blending :	
Amount of LCGO used in Motorin	196.7936
Amount of LCGO used in Motorin	386.46
Amount of Diesel used in Motorin	0
Amount of Kero-Mot used in Motorin	0
Amount of Die used in Motorin	7392.2241
Motorin	7975.478
F.Oil Blending :	
Amount of VRSD used in Ozel Kal Yak	2156.1899
Amount of VRSD used in Ozel Kal Yak	2934.6641
Amount of F.Oil used in Ozel Kal Yak	274.6055
Amount of F.Oil used in Ozel Kal Yak	0
Amount of VRSD used in F.Oil 6	0
Amount of VRSD used in F.Oil 6	0
Amount of F.Oil used in F.Oil 6	89.2445
Amount of F.Oil used in F.Oil 6	10.7555
Amount of VRSD used in Burner F.Oil	0
Amount of VRSD used in Burner F.Oil	0
Amount of F.Oil used in Burner F.Oil	0
Amount of F.Oil used in Burner F.Oil	100
Ozel Kal Yak	5365.46
F.Oil 6	100
Burner F.Oil	100
7 LPG :	
Sulphur	17.372

FG	8.686
LPG	842.5397
7 FCC :	
WCN	596.4588
LCGO	196.7936
LPG	317.409
FG	62.1311
F.Oil	110.7555
4 FCC :	
WCN	765.3199
LCGO	386.46
LPG	236.17
FG	48.45
F.Oil	363.85
6 D :	
Kero	0
FG	0
36 D :	
Diesel	0
FG	0
3 D :	
Kero	300
FG	10.3983
6 P :	
Plat	0
LPG	0
FG	0
36 P :	
Plat	1295.7271
LPG	44.9385
FG	157.2848

6 U :	
LSR	0
3 U :	
LSR	100
36 U :	
HSR	1497.95
Kero Tank :	
Kero	310.3983
HSR Tank :	
HSR	1597.95
LSR Tank :	
LSR	2339.7219
HVGO Tank :	
HVGO	3350.677
2 V :	
LVGO	754.2857
HVGO	1508.571
VRSD	2156.1899
5 V :	
LVGO	2223.23
HVGO	1842.105
VRSD	2934.6641
5 HP :	
Diesel	1397.459
Kero	1460.98
HSR	1079.855
LSR	1486.3879
ATSR	7000
FG	25.4083

LPG	215.971
2 HP :	
Diesel	914.2857
Kero	952.381
HSR	518.0953
LSR	853.3333
FG	15.2381
ATSR	4000
LPG	99.0476
CRUDE OIL TANK :	
Crude Oil iran light	0
Crude Oil sarir	0
Crude Oil essider	20323.2207
Crude Oil essider to 2 HP	7619.0479
Crude Oil sarir to 2 HP	0
Crude Oil iran light to 2 HP	0
Crude Oil essider to 5 HP	12704.1699
Crude Oil sarir to 5 HP	0
Crude Oil iran light to 5 HP	0

REFERENCES

[1] Bodington, C.E., Baker, T.E., "A History of Mathematical Programming in the Petroleum Industry", *Interfaces*, Vol. 20, No. 4, pp. 117-127, 1990.

[2] Symonds, G. H., "Linear Programming Solves Refining and Blending Problems", *Industrial and Engineering Chemistry*, Vol. 48, No. 3, pp. 394-401, 1956.

[3] Charnes, A., Cooper W. W., and Mellon, B., "Blending aviation gasoline-A study in programming interdependent activities in an integrated oil company", *Econometrica*, Vol. 20, No. 2, pp. 135-139, 1952.

[4] Manne, Alan, " A linear programming model of the US petroleum refining industry", *Econometrica*, Vol. 26, No. 1, pp. 67-106, 1958.

[5] Kawaratani, T. K., Ullman, R. J., Dantzig, G. B., "Computing Tetraethyllead Requirements in a Linear Programming Format", *Operations Research*, Vol. 8, pp. 24-29, February 1960.

[6] Baker, T. E., Lasdon, L. S., "Successive Linear Programming a Exxon", *Management Science*, Vol. 31, No. 3, pp. 264-274, 1985.

[7] Klingman, D., Phillips, N., Steiger, D., Wirth, R., Padman, R., Krishnan, R., "An Optimization Based Integrated Short-Term Refined Petroleum Product Planning System", *Management Science*, Vol. 33, No. 7, pp. 813-829, 1987.

[8] Klingman, D., Padman, R., Phillips, N., "Intelligent Decision Support Systems: A Unique Application in the Petroleum Industry", *Annals of Operations Research*, Vol. 12, pp. 277-283, 1988.

[9] Dewitt, C. W., Lasdon, L. S., Waren, A. D., Brenner, D. A., Melhem, S. A., "OMEGA: An Improved Gasoline Blending System for Texaco", *Interfaces*, Vol. 19, pp. 85-101, *1989*.

[10] Uhlmann, A., "Linear Programming on a Micro Computer: An Application in Refinery Modeling", *European Journal of Operations Research*, Vol. 35, pp. 321-327, 1988.

[11] Koşal, H., "A methodology for Production Planning in a Refinery", Unpublished M. Sc. Thesis, Department of Industrial Engineering, M.E.T.U., Ankara, 1981.

[12] Kavrakoğlu, İ., Or, İ., Eyler, M. A., Kaylan, A. R., Doğu, G., "TÜPRAŞ İzmit Rafinerisinin Üretim Planlamasının Geliştirilmesi Projesi Son Raporu", Mühendislik Fakültesi, Boğaziçi Üniversitesi, İstanbul, 1986.

[13] Buchanan, J. E., Garven, S. C., Geniş, O., Shapiro, J. F., Singhal, V., Thomas, J. M., Torpiş, S. , "A Multi-Refinery, Multi-Period Modeling System for the Turkish Petroleum Refining Industry", *Interfaces*, Vol. 20, No. 4, pp. 48-60, 1990.

[14] Gürkan, T., Kartal, N., "Model for the Development of the Turkish Petrochemical Industry", *Engineering Costs and Production Economics*, Vol. 18, pp. 145-157, 1989.

[15] Greenberg, H. J., "A Bibliography for the Development of an Intelligent Mathematical Programming System", *ORSA CSTS NewsLetter*, Vol. 15, No. 1, pp. 21-37, 1994.

[16] Krishnan, Ramayya, "Model Management: Survey, Future Directions and a Bibliography", *ORSA CSTS Newsletter*, Vol. 14, No. 1, pp. 7-22, 1993.

[17] Geoffrion, A. M., "An Introduction to Structured Modeling", *Management Science*, Vol. 33, No. 5, pp. 547-588, 1987.

[18] Geoffrion, A. M., "The Formal Aspects of Structured Modeling", *Operations Research*, Vol. 37, No. 1, pp. 30-51, 1989.

[19] Geoffrion, A. M., "Structured Modeling: Survey and Future Research Directions", *ORSA CSTS NewsLetter*, Vol. 15, No. 1, pp. 1-20, 1994.

[20] Geoffrion, A. M., "Computer Based Modeling Environments", *European Journal of Operations Research*, Vol. 41, pp. 33-43, 1989.

[21] Murphy, F. H., Stohr, E. A., Asthana, A., "Representation Schemes for Linear Programming Models", *Management Science*, Vol. 38, No. 7, pp. 964-991, 1992.

[22] Brooke, A., Kendrick, D., Meeraus, A., *GAMS, Release 2.25, A User's Guide*, The Scientific Press, 1991.

[23] Fourer, R., Gay, D. M., Kernighan, B. W., *AMPL A Modeling Language for Mathematical Programming*, The Scientific Press, 1993.

[24] Fourer, R., Gay, D. M., Kerninghan, B. W., "A Modeling Language for Mathematical Programming", *Management Science*, Vol. 36, No. 5, pp. 519-555, 1990.

[25] Schrage, L., *LINDO, Release 5.0*, The Scientific Press, 1991.

[26] Choobineh, Joobin, "SQLMP: A Data Sublanguage for Representation and Formulation of Linear Mathematical Models", *ORSA Journal on Computing*, Vol. 3, No. 4, pp. 358-375, 1991.

[27] Geoffrion, A. M.,"The SML Language for Structured Modeling: Levels 1 and 2", *Operations Research*, Vol. 40, No. 1, pp. 32-57, 1992.

[28] Geoffrion, A. M., "The SML Language for Structured Modeling: Levels 3 and 4", *Operations Research*, Vol. 40, No. 1, pp. 58-75, 1992.

[29] Greenberg, H. J., Murphy, F. H., "A Comparison of Mathematical Programming Systems", *Annals of Operations Research*, Vol. 38, pp. 177-238, 1992.

[30] MA, Pai-chun, Murphy, F. H., Stohr, E. A., "Representing Knowledge About Linear Programming Formulation", *Annals of Operations Research*, Vol. 21, pp. 149-172, 1989.

[31] MA, Pai-chun, "An Intelligent Approach Towards Formulating Linear Programs", Ph.D. Dissertation, New York University, 1988.

[32] Jones, C. V., "User Interfaces", in: Coffman et al. (ed.), *Computing, Handbooks in OR&MS*, Vol. 3, pp. 603-668, North-Holland, Amsterdam, 1992.

[33] Jones, C. V., "Visualization and Optimization", *ORSA Journal on Computing*, Vol. 6, No. 3, pp. 221-257, 1994.

[34] Bell, P. C., " Visual Interactive Modeling: The Past, The Present and the Prospects", *European Journal of Operations Research*, Vol. 54, pp. 274-286, 1991.

[35] Bright, J. G., Johnston, K. J., "Whither VIM - A Developers View", *European Journal of Operations Research*, Vol. 54, pp. 357-362, 1991.

[36] Hurrison, R. D., "Intelligent Visual Interactive Modelling", *European Journal of Operations Research*, Vol. 54, pp. 349-356, 1991.

[37] Thomas, D., "What is an Object", *Byte*, pp. 231-240, March 1989.

[38] Coad, P., Yourdon, E., *Object Oriented Analysis*, Yourdon Press, 1991.

[39] Coad, P., Yourdon, E., *Object Oriented Design*, Yourdon Press, 1991.

[40] Dodani, M. H., Hughes, C. E., Moshell, J. M., "Separation of Powers", *Byte*, pp. 255-262, March 1989.

[41] *ObjectWindows for C++, User's Guide*, Borland International Inc., 1991.

[42] Piela P.C., Epperly, T. G., Westerberg, K. M., Westerberg, A. W., "ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language", *Computers Chemical Engineering*, Vol. 15, No. 1, pp. 53-72, 1991.

[43] Altınel, İ. K., Birgören, B., Draman, M., "TÜPRAŞ İzmit Rafinerisi için bir Doğrusal Programlama Modeli", Research Paper Series No: FBE-IE-06/93-08, Department of Industrial Engineering, Boğaziçi University, İstanbul, 1992.

[44] Altınel, İ. K., Birgören, B., Draman, M., "Bir Türk Rafinerisi için Doğrusal Programlama Modeli", Yöneylem Araştırması ve Endüstri Mühendisliği XV. Ulusal Kongresi Bildiriler Kitabı, Boğaziçi University, İstanbul, 1993.

[45] Birgören, B., Draman, M., Altınel, İ. K., "Bir Petrol Rafinerisinde Üretimin Eniyilenmesi için bir Görsel Etkileşimli Modelleme Sistemi", Yöneylem Araştırması

ve Endüstri Mühendisliği XVI. Ulusal Kongresi Bildiriler Kitabı, Bilkent University, Ankara, 1994 (to appear).

[46] *Borland C++, Programmer's Guide, Version 3.1*, Borland International Inc., 1992.