DESIGN AND IMPLEMENTATION OF GRAPHICAL USER INTERFACES FOR COMPLEX SIMULATIONS

FÁTIMA ISABEL CARIA CANELAS PAIS GENSBERG

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

February 1995

THE UNIVERSITY OF ASTON IN BIRMINGHAM

DESIGN AND IMPLEMENTATION OF GRAPHICAL USER INTERFACES FOR COMPLEX SIMULATIONS

FÁTIMA ISABEL CARIA CANELAS PAIS GENSBERG
Doctor of Philosophy
1995

Summary

The development of increasingly powerful computers, which has enabled the use of windowing software, has also opened the way for the computer study, via simulation, of very complex physical systems. In this study, the main issues related to the implementation of interactive simulations of complex systems are identified and discussed.

Most existing simulators are closed in the sense that there is no access to the source code and, even if it were available, adaptation to interaction with other systems would require extensive code re-writing. This work aims to increase the flexibility of such software by developing a set of object-oriented simulation classes, which can be extended, by subclassing, at any level, i.e., at the problem domain, presentation or interaction levels. A strategy, which involves the use of an object-oriented framework, concurrent execution of several simulation modules, use of a networked windowing system and the re-use of existing software written in procedural languages, is proposed.

A prototype tool which combines these techniques has been implemented and is presented. It allows the on-line definition of the configuration of the physical system and generates the appropriate graphical user interface. Simulation routines have been developed for the chemical recovery cycle of a paper pulp mill. The application, by creation of new classes, of the prototype to the interactive simulation of this physical system is described. Besides providing visual feedback, the resulting graphical user interface greatly simplifies the interaction with this set of simulation modules.

This study shows that considerable benefits can be obtained by application of computer science concepts to the engineering domain, by helping domain experts to tailor interactive tools to suit their needs.

Keywords

Design Framework ⋆ Mathematical Modelling ⋆ Concurrent Simulation ⋆ Chemical Recovery Cycle

To Karl, for his endless patience and encouragement during the course of this work.

I am grateful to Dr. Brian Gay, my supervisor, for his guidance and encouragement.

I would also like to thank Dr. António Portugal, from the Department of Chemical Engineering of the University of Coimbra, Portugal, for advice on chemical process simulation.

The financial support provided by the Commission of the European Communities, under the Science Program grant contract No.B-SC1/900596, is gratefully acknowledged.

Finally, I would like to thank my husband and parents for their support during this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scope and Aims of Research

The development of increasingly powerful computers, which has enabled the use of windowing software, has also opened the way for the computer study, via simulation, of very complex physical systems. In chemical engineering, for example, simulation is typically used for steady-state calculations, which include process flowsheeting and design, and transient-state calculations, i.e. the study of the dynamic response of the process to disturbances, used in tuning control systems or determining the parameters of the process model. If the analysis of process dynamics under conditions of start-up, grade changes, emergency outages and shutdowns is desired, software for dynamic simulation is necessary (Shewchuk *et al.*, 1991). Dynamic simulators are also being increasingly used to support employee education and training (Ragnemalm, 1993). The goal of training is to increase the operator's understanding of the process, for which it is necessary to design and implement tools that can be used by operators to improve their skills in diagnosing faults and solving operational problems. The migration of simulation tools from "simulation experts" into the hands of process engineers and operators will be accelerated by the availability of effective user interfaces (Shewchuk *et al.*, 1991).

Graphical User Interfaces (GUI's) have taken over user interface development in the last few years, and a number of GUI's have recently been devised for systems that range from genetics applications (Drury *et al.*, 1992) to economic analysis (Jensen and King, 1992) and spectral data processing (King and Horlick, 1992). However, although GUI's have been developed for a wide range of applications, few GUI's are related to on-line scientific simulations, i.e., the computer simulation of physical systems of arbitrary dimension and complexity, described by mathematical models whose solution requires intensive application of specialized numerical methods. Most problems found in engineering belong to this class and many are still solved by batch runs. Advantage is not taken of interactivity and there is a considerable waste of time due to the migration of result files to different environments, and, in most cases, preparation of the results for graphical representation.

On the other hand, although the application of object-oriented concepts to software development has proven to yield significant benefits, such as increased modularity and reusability, only recently have these concepts found their way to general process engineering computations (Stephanopoulos *et al.*, 1987). Traditional procedural simulation programs take the form of calls of lengthy procedures of numerical methods, required for the solution of the mathematical models that describe the physical systems. Most of these procedures are available in standard libraries written in high-level procedural languages such as Fortran77 or C. Although considerable research efforts are now being directed towards the problem of implementing object-oriented versions of advanced numerical methods, full general-use libraries of OO implementations of numerical methods do not yet exist, which is the main reason why few complex systems are at present simulated using these concepts.

Additionally, very complex problems may not be suitable for on-line simulation due to the execution time, which can lead to unacceptably long response times. However, parallel processing techniques may significantly reduce execution times; the development of parallel algorithms for the solution of problems typically solved sequentially is currently a major field of research. Software has also been recently developed that enables inter-process communication and the use of networked machines, such as UNIX workstations, as a single distributed memory machine thus making it possible to implement parallel processing techniques on commonly available computer networks.

Finally, if the interactive application is to be developed in a structured way, the graphical user interface must be kept separate from the domain-specific component. GUI toolkits (collections of graphical objects, or widgets, such as windows, menus, scrollbars, etc.) alone do not provide a framework for the application, which tends to become a large agglomeration of intricately related widgets. The MVC (Model-View-Controller) framework originally developed in Smalltalk (Goldberg and Robson, 1983; Goldberg, 1984) is ideally suited to the development of this kind of application. The model objects map directly to the components of the physical system, whereas the view objects represent the type of display desired for the output of the models, and the controller objects enable the user to interact with the models and the views. As an alternative to Smalltalk, client-server based windowing systems like X Window (Scheifler and Gettys, 1986) provide more flexibility (and in most cases a greater execution speed), allowing full advantage to be taken of heterogeneous networks of computers.

Clearly, a number of different issues are involved in the development of interactive software for complex simulations; an effective approach must merge the different solutions found for each aspect into a coherent whole. This problem therefore involves:

1. development of a strategy to define a physical system on-line, using predefined object classes. This implies creation and deletion of objects at run-time, as well as a methodology to connect them mathematically, and a tool to enable the user to perform these operations graphically;

2. means to re-use existing code, encapsulated by an object-oriented environment;

3. use of parallel processing techniques, either at the level of the solution algorithms, or, at a coarser level, assigning each object to a (possibly different) processor;

4. use of a windowing environment that also allows full advantage to be taken of the available machines in the network;

5. an *integrated environment* to organize the numerical data, perform and control the simulation, and organize and display its results. A framework must therefore be provided, so that a clear separation is retained between the GUI and the domain-specific component.

The basic aim of this research is to identify, discuss, and propose solutions to the main problems related to the implementation of interactive simulations of complex systems. In the chemical process engineering domain, for example, the model representations and modelling tools provided by established process simulators do not sufficiently address the practitioner's needs especially if detailed and non-standard models are required (Marquardt, 1992). The current trend of research aims to explore the benefits of an object-oriented approach together with the existence of powerful graphical user interfaces (Stephanopoulos *et al.*, 1987; Bār and Zeitz, 1990; Shaw, 1992; Zobel and Lee, 1992), as an alternative to the use of conventional hardware and software environments (e.g. ASPEN, 1988). KEE (Knowledge Engineering Environment) (Fikes and Kehler, 1985), a LISP-based expert system building tool, has been used by Stephanopoulos *et al.* and Bār and Zeitz to describe the domain knowledge base (e.g. process units, control loop components, etc.) in the first case, and to implement the user interface in the second. Stephanopoulos *et al.* used Flavors (Symbolics Inc., 1986) for the development of the user interface, together with graphical objects available in SYMBOLICS 3640 and 3650 computers. Bār and Zeitz also used a SYMBOLICS LISP workstation with high resolution bitmapped graphics for the graphical user interface, but implemented the numerically-intensive computations in Fortran77. Another approach (Shaw, 1992), although claiming to be object-oriented, uses C for the graphical user interface, and Fortran or C for the numerical calculations. Finally, Zobel and Lee used C and Borland C++ (Borland, 1990), together with Microsoft Windows constructs, for the graphical interface and to define the object base. From the above, it is obvious that the current software *state-of-the-art* provides many alternatives, most of which have not been fully explored yet.

In this work, a totally open architecture is aimed at, either at the level of the domain-specific component (by adding new types of model objects or refining existing ones), or at the level of the graphical user interface (e.g., by providing different types of graphical representation of the variables or even by changing the user interface, or the interaction techniques, offered by the simulator itself). This degree of openness can only be achieved in a structured way if a clear, uniform, fully object-oriented environment such as the one defined by frameworks like MVC is imposed. A possible solution is to develop a set of object-oriented class libraries, which are easy to extend or modify, rather than developing a self-contained product. The possibility of inserting user-defined simulation modules in a straightforward manner is offered in most of the latest simulators and is especially important if interaction is desired with non-standard models, which will not normally be present in domain-specific libraries. In this work, one step beyond is taken, since the type of display and interaction must also be flexible and easy to change. As mentioned before, toolkits *per se* are insufficient for the creation of structured large applications. Since special emphasis is placed on the graphical user interface, a means to make a more effective use of toolkits must be provided. Rather than developing a specific simulator for a specific problem, this work therefore aims to provide the *tools* for interactive simulator development.

Depending on the complexity of the process model, response time will in most cases become a problem. The possibility of implementing parallel processing techniques and using them on-line (which has not been referred to in any of the previous simulators mentioned above), in order to make an effective utilization of the network resources and minimize response times, is to be studied. No limitations must be imposed on the type of paradigm used, since, for maximum efficiency, this is problem-dependent.

The selection of the software used has major consequences on the degree of portability of the resulting system. In order to achieve the highest possible degree of portability, the programming languages and graphics systems must be standard; the use of graphical objects available only in a specific machine is a severe restriction. X Window has become a *de facto* standard for Unix workstations, and can also be used in other (e.g. VMS) machines. C++ and Fortran77 compilers are almost universally available, justifying their use in this work.

In order to exemplify the approach taken, class libraries must be developed and a tool implemented using these classes which, in addition to enabling the graphical definition of the model for the physical system, also allows the user to interact with the simulation. This tool must take into account the dynamic nature of the application (the model for the physical system is to be defined and/or altered during run-time) and enable the incorporation of concurrency/parallel processing techniques if desired (remote processes may be created and destroyed at run-time). Finally, the prototype developed must be tested by application to an example of a real system, representative of the problems faced.

The example selected was the chemical recovery cycle of paper pulp plants, which has gained renewed interest in recent years due to both environmental considerations and the high price of fuel. This involves the development and implementation of detailed mathematical models for the units that comprise the cycle; no such collection of models is available in the literature. Although the prototype developed must be domain-independent, its application to this case study is an interdisciplinary approach that enables the identification of the potential benefits of the techniques listed above when applied to engineering areas.

1.2 Thesis Structure

This work is composed of eight chapters and five appendices.

This chapter (Chapter 1) provides an overview of the areas involved and the aims of this research.

Chapter 2 introduces the object-oriented paradigm and its basic concepts, namely the existence of objects, classes, messages, abstraction, encapsulation, polymorphism, inheritance and delegation. The basic methodology for object-oriented design is reviewed, and the most commonly used object-oriented programming languages are listed. A brief reference is also made to the exception-handling mechanisms provided by object-oriented languages.

Chapter 3 provides an overview of the main aspects involved in Human-Computer Interaction (HCI). This is a very extensive field and only the most important aspects are presented, covering both computing and human-factors aspects. These include the analysis of the type of interaction that the GUI offers to the user and a list of the main issues usually referenced in human-factors guidelines. User interface design is briefly outlined and mention is made of the available toolkits and graphics systems, with special reference to the X Window windowing system and the XView toolkit. The requirement for a framework is stressed and, finally, several techniques for the evaluation of the usability of the interface are outlined.

Chapter 4 presents the main issues related to currently available concurrent software and hardware systems. The client/server paradigm is described, with reference to the Remote Procedure Call (RPC) libraries and to the client/server structure of X Window. An introduction to different types of parallel computers is made, as well as to the more recent message-passing languages (namely, actor-based languages and concurrent object-oriented programming languages). Finally, a novel software system (PVM) which enables inter-process communication without enforcing any type of paradigm is introduced, and the need for effective operating system support for concurrent applications is stressed.

Chapter 5 presents an introduction to mathematical modelling of physical systems and to the existing simulation languages. The main issues related to object-oriented interactive simulation are identified and discussed. The basic principles used in the prototype developed in this work are described. These include the decomposition of the overall physical system into sub-models, the use of modules written in procedural languages for the computationally-intensive sections, and the development of a framework adequate for the management of a (potentially large) number of entities in a structured way. Finally, two different distribution approaches, based on RPC and PVM (Parallel Virtual Machine), implemented during different iterations of the prototype, are described and discussed.

Chapter 6 describes in greater detail the standard class libraries developed in this work. It also presents the strategy used for the control of the execution status of the remote processes created by the interface. The methodology used to join the C++ classes developed with the underlying X Window and XView layers, and the problems encountered, are listed. Finally, the tool developed for the on-line definition of the physical system and the management of the simulation is described.

Chapter 7 presents a case study, namely the chemical recovery cycle of a paper pulp plant. A brief description of the industrial process is made, with special reference to the complexity of the mathematical models developed. The specification of the functions that the interface must perform for this case is outlined and the extension of the base class library is described. Quantitative results are presented that enable the visualization of:

— typical physical behaviour of the industrial units involved;

— the effect of the decomposition of the overall model on the accuracy of the results;

— the effect of the concurrent execution of simulation processes on the response time.

Finally, Chapter 8 presents a summary of the work developed and the conclusions that may be drawn from this study. It also suggests further avenues of research which may either enhance the efficiency of the prototype or provide alternative methods of solution for the problems posed.

The mathematical models developed for the simulation of the chemical recovery cycle of a paper pulp mill are presented in Appendices A, B, C, and D. Appendix A presents both the dynamic and steady-state models for the lime kiln, the most energy-consuming unit in the cycle. Appendix B presents a set of models developed for the causticizing units. They include a detailed model for the solid particles, determination of the steady-state of the causticizing reactors, and a dynamic model for the causticizing battery. The dynamic model for the white liquor clarifier is presented in Appendix C and, finally, Appendix D includes an approximate model for the lime mud filter. Appendix E lists all the files used by the prototype as well as the directory structure in which these files are arranged.

Chapter 2

Object-oriented Programming

2.1 Introduction

The evolution of the technological means available, since Charles Babbage's "Analytical Engine", for the creation of computing devices, has been matched by an evolution in both the programming languages and the design principles applied. Among these, Object-Oriented Programming (OOP) offers a programming paradigm which is more efficient than any before to handle applications which are ever increasing in complexity (Yip and Dessey, 1991). Object-oriented technology results in significant re-use of code, design, and analysis; reduced number of development cycles; and a more efficient way of processing requirements (Kamath *et al.*, 1993). This is due to the fact that object-oriented analysis fits right into the problem domain (Coad, 1991). It is possible to keep the problem domain organizational framework intact, with the consequence that the relationship between analysis and design and between design and programming is evident since all of them are organized like the problem domain itself (Coad, 1991). The exact combination of attributes which must be present in an object-oriented language has not been universally agreed upon and, furthermore, the expression "object-oriented" has been used with several meanings (Gray and Mohamed, 1990). Although different authors tend to emphasize different aspects, the basic features of object-oriented programming together with the associated terminology will be introduced next.

2.2 Basic Concepts

The concepts involved in object-oriented programming are commonly referred to as abstraction, encapsulation (or information hiding), polymorphism, and inheritance or delegation. As a practical means to implement these concepts, an object-oriented language must, among other requirements, be class *and* object-based. Some languages are class or object-based only; these languages cannot be considered truly object-oriented. A brief definition of object-based, class-based and object-oriented languages is given in section 2.2.3.

In the object-oriented programming paradigm, the attention is focussed on objects (Pokkunuri, 1989). An object is a unit, or entity, characterized by data attributes, stored in variables, and procedure attributes (methods). It executes one of its methods as a result of receiving a message from another object (client); an object interacts with other objects and the outside world through messages which it sends and receives. Methods can be divided into two categories (Meyer, 1993): commands and queries. A query returns some information about an object, whereas a command may modify the state of the object. Some authors state that messages should have the same status as objects, since "an object-oriented system is a collection of objects that accomplish prescribed activities by sending messages to each other" (Dean, 1991). Behavioural compositions, i.e., groups of interdependent objects cooperating to accomplish tasks, are an important feature of object-oriented systems (Helm *et al.*, 1990). The fewer assumptions an object makes about its environment (and vice-versa) the easier it becomes to translate it into a different environment (Wilson, 1990). In this context, "environment" means the rest of the program, or related objects in the system.

It can sometimes be useful to think of objects as people. The anthropomorphization of objects helps assimilate the concept that objects, like people, can be thought of as sentient to some degree. They know things about their own state, how to perform certain tasks, and how to communicate with other objects (Crenshaw, 1991). Failure to fully use these concepts results in thinking of objects as passive entities, not capable of initiative, which in turn leads to not taking full advantage of the object-oriented approach.

In object-oriented languages, objects are organized in classes, and the data structures and methods are not described individually for each object. Rather, for a group of similar objects, the description is stored in the definition of the class that the objects belong to. An object of a certain class is referred to as an instance of that class; its internal data are referred to as the instance variables. A subclass must be created whenever the behaviour of a method needs to be changed, a new method must be added, or more instance variables must be added (Think C, 1991). Programming in OOP therefore consists basically of identifying entities and concepts in an application and modelling them using classes. In class-based languages, the class is a mould, used to manufacture objects (Cointe, 1987); in most languages, like C++, a class itself is not an object. Languages like Loops and Smalltalk-80 introduced the concept of metaclass (i.e., the class of a class) to provide greater abstraction, allowing the description of a class by another class. From the user's point of view, metaclasses define the *class methods*, which make it possible to send messages to classes, and the instance variables at the class level, which enable the user to parametrize classes (Briot and Cointe, 1989).

2.2.1 Abstraction and Encapsulation

Abstraction can be defined as "extracting essential properties of a concept" (Ege, 1992), or "representing the essential features of something without including background or inessential detail" (Graham, 1994). Examples of abstraction in procedural programming are the concepts of functions and primitive data types. Rather than considering the specific details of a specific function, all functions can be grouped together in a general category, since they all share the same basic properties. A class is an abstraction which describes the general behaviour of a group of similar objects; it is an implementation of an *abstract data type*. In practical terms, *data abstraction* is the separation of an object's implementation from its abstract behaviour (Purchase and Winder, 1991). It implies that the sender of a message to an object need not concern itself with how the request is satisfied; that is the exclusive responsibility of the receiving object. It is therefore possible to create and use an object without being directly involved in its internal structure. The programmer can concentrate more on what the interactions should be than on how they will be performed, and the design effort is directed towards problem specification rather than implementation details (Pokkunuri, 1989). Abstract data types, unlike primitive data types, may be defined by the application developer, rather than by the designer of the language used (Graham, 1994). Data abstraction can be viewed as a generalization technique and is usually accompanied by encapsulation (Coplien, 1992).

Encapsulation is the process by which a client is restricted, in its interaction with another object, to the external interface of that object (i.e., the available *methods*)(Snyder, 1986). Encapsulation can be viewed as an information hiding technique (Coplien, 1992). The external interface of a class serves as a contract between the class and its clients, that is, the objects that communicate with instances of that class. If the clients depend only on the external interface, the class can be reimplemented without affecting any clients, as long as the new implementation supports the same or a compatible interface. This assures designers that compatible changes can be made safely, which facilitates program evolution and maintenance (Snyder, 1986). Furthermore, because each object has a limited view of the other objects and a limited access to the other objects' data, accidental corruption of data is minimized.

Cook and Daniels (1992) use the metaphor of the boiled egg, where the yolk represents the data structures, the white the implementations of the methods and the shell the interface or protocol (see Fig.2.1). An object sends a message as part of the actions implemented in its methods. Therefore, when a message is sent, it originates in the white. It then collects data from the yolk and is sent to the shell of the receiving object (Graham, 1994). Note that this is a simplistic metaphor, ideally suited for languages like Smalltalk, where all the instance variables are hidden and all the methods are exposed. C++, for example, offers a wider choice of encapsulating measures that cannot be represented in such a simple way.

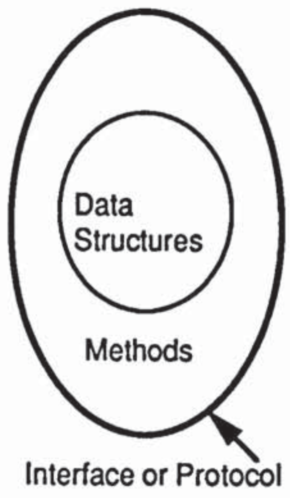


Figure 2.1: The egg metaphor.

2.2.2 Polymorphism, Inheritance and Delegation

The concepts of polymorphism and inheritance are again strongly related. Inheritance allows a new class to be built upon an existing one. The new class is called a subclass, derived class or child class of the existing one (superclass or parent class), and inherits all the general characteristics of the superclass (Yip and Dessey, 1991). The subclasses can add new data and behaviour properties, thus becoming more specialized. The ability to re-define an inherited method is called overriding. This is the actual mechanism by which classes can replace methods inherited from the superclass with their own. As an example, objects of the class LinearRandom (Mitchell, 1993) generate random numbers. The variable Seed, which initializes the numerical process, is hidden, and the method SetSeed is used to manipulate it. The method NextRandom generates the next random number, according to a linear algorithm. Class GaussianRandom inherits from class LinearRandom (see Fig.2.2). This means that it contains some parts of a LinearRandom; however, the algorithm for generating the random numbers must be different, namely, it must be the average of 12 numbers generated by the LinearRandom algorithm. In order to achieve this, method NextRandom has been overridden (re-defined) by class GaussianRandom.

In most OOP languages, the binding of the operator to a particular operation takes place at run-time (*dynamic* or *late binding*); the actual method to

Figure 2.2: Class GaussianRandom is a subclass of class LinearRandom. (From Mitchell, 1993).

call is determined during execution. This means that the same message takes different forms — hence the term polymorphism, from the Greek "many shapes" — depending on the class of the object that receives it. Polymorphism is based on the assumption that the type (or class) of a variable need not match the type (or class) of the object that the variable refers to; a variable declared to be of a certain class may refer to objects of that class, or objects of any subclass. Functions that would be coded as large case statements in conventional languages (one function only, which would perform different operations depending on the value of the arguments) are distributed over classes, with a consequent improvement in the structure of the program (Grogono and Bennett, 1989).

Since new classes can be created which take full advantage of existing ones, yet without changing them, inheritance promotes code reusability, allowing the programmer to draw from and extend large amounts of existing code. It is the object-oriented concept that contributes most to the productivity increase attained by object-oriented languages (Ege, 1992). Also, the modularity of the program is increased by the class description. It acts as the module that allows the separation of the program into smaller sections.

The concept of *delegation*, although sometimes used in a similar context as inheritance, is however subtly different (Mullin, 1989). *Delegation* is not directly supported by C++ but can be found in other OO languages. If *delegation* is supported, a member function applied to one object can be transparently forwarded to another object, and data member references can be similarly deferred from one object to another. The second object's member function executes in the context of the first object. Delegation is usually used in classless languages; since class inheritance does not exist, this mechanism avoids duplication of code (Graham, 1994). The equivalent of delegation in class-based languages like C++ is inheritance. Delegation means more than inheritance: its primary function is to distribute the workload of an object to more specialized objects linked to it (Mullin, 1989).

Other concepts such as persistence and concurrency are also referred to in an object-oriented context. Persistence can be defined as "the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created)" (Booch, 1991). Concurrency can be defined as "the property that distinguishes an active object from one that is not active" (Booch, 1991). Whereas persistence is not fundamental in this work and will not be referred to further, concurrency is an important issue and will be addressed in greater detail in Chapters 4 and 5.

2.2.3 Object and Class-based Languages

There are several programming languages which, while not strictly obeying all the object-orientation requirements, provide some object-oriented features. Although this classification is not universal, such languages are usually categorised into object and class-based languages.

An *object-based* programming language supports encapsulation and object identity (in the sense that each object has a unique identifier) (Graham, 1994). However, set abstraction (existence of classes) is not supported. Objects do not belong to classes, and there is no support for inheritance. Ada is an example of such a language.

Class-based languages, such as CLU, include all the features of object-based ones, and in addition support the existence of classes from which objects are instantiated. They do not however support inheritance between classes.

Finally, object-oriented languages merge the characteristics of both object and class-based systems (Graham, 1994). In addition, they also support inheritance between instances and classes, and classes and classes, i.e., both instances and classes inherit the methods and attributes of the class(es) they belong to. C++ and Smalltalk are examples of object-oriented languages. For more on language classification, see Wegner (1987).

Examples of object-based alternatives to the class model are the actor model (Agha, 1986) or the prototype metaphor for object creation as in Self (Ungar and Smith, 1987), which allow other organizations of knowledge. In Self, some objects are designated by convention as exemplars or prototypes, from which other objects with similar properties are "cloned". After being cloned, they may be modified and fine-tuned by changing properties associated with classes or objects in class-based languages.

2.3 Object-oriented Software Life Cycle

Classical systems analysis and design techniques include the requirements analysis, requirements specification, system design, detailed design, and implementation (Ince, 1991). Each of these stages produces some kind of output which is then fed to the next stage. In object-oriented programming, the fundamental decomposition criterion is the search for objects or classes that model problem domain entities or concepts (Caromel, 1993; Mitchell, 1993). If classical techniques are applied to OOP, the *system design* stage can thus be split into a) object identification and b) definition of the interface of each class and of the dependencies among classes. Detailed design corresponds to the specification of each class (Caromel, 1993; Mitchell, 1993), i.e., which variables are required and which functions.

This model for the software life cycle, in which each stage must be complete before the next one starts, is known as the *waterfall model* (Sommerville, 1992). Although improvements to this model include feedback loops between stages, these are still confined to successive stages (Fong, 1993). The waterfall model does not account for iteration in software development and is therefore especially inadequate to describe the development of object-oriented software, which places a strong emphasis on software re-use and is fundamentally incremental and iterative in nature. Moreover, as Fong (1993) points out, the waterfall model relies on the elaboration of fully finished documents during the analysis and specification stages before design can proceed. Although this can work well if a fixed set of requirements is known from the beginning, it does not allow for the uncertainty in the requirements which exists in many projects.

In practice, these sharp boundaries between stages are not present during the development of object-oriented software, since interest is always focussed on the same entities, *objects*, throughout analysis and design. Information derived during the analysis stage can be directly used during design, instead of providing a mere starting point for a separate stage. Alternatives to the waterfall model are the *fountain model* (Henderson-Sellers and Edwards, 1990), the *cluster model* (Meyer, 1989) and the *spiral model* (Wasserman and Pircher, 1991). In the fountain model, the top of the "fountain" corresponds to software use, from which it can fall to previous stages due to maintenance needs or to further development. In the stages before it reaches the top, however, it can always fall to the previous stage and start climbing again from there (hence the name of the model). The *cluster model* groups related classes into clusters; each cluster is fairly independent from the others and therefore the fountain model can be applied, concurrently, to each cluster of classes. Although alterations in the requirements may cause major changes or even dismissal of a cluster, the rest of the software will not be greatly affected since the system will not be put together until later (when all the surviving and refined clusters have been implemented). Finally, the *spiral model* as presented by Wasserman and Pircher (1991) is conceptually similar to the fountain model since it emphasises the iterative nature of the software life cycle and considers that no sharp boundaries exist. Since it is dedicated to object-oriented programming, each stage corresponds to a specific function in OO analysis and design (e.g., *Find Classes/Objects*; *Find Methods/Behaviours*; *Define Classes*, etc.).

Superimposed on each of these stages, other activities must be continuously applied, to ensure that the system carries out the specified tasks correctly (*verification*), and that it meets user requirements (*validation*) (Ince, 1991). In the case of graphical user interfaces, another important activity is *evaluation* (see Chapter 3) which includes the rating of the subjective satisfaction experienced by the user when s/he interacts with the system. These activities imperatively require that software be developed in an iterative manner, i.e, the waterfall model is inadequate for the development of this kind of software system.

Fong (1993) and Graham (1994) present, in greater detail, an overview of object-oriented design methods; see also Coad and Yourdon (1991) for object-oriented analysis.

2.4 Overview of Object-oriented Languages

The flexibility and support offered by the latest high-level procedural languages such as Fortran, ALGOL, C or Pascal is considerable when compared to machine code or assembly language (Pokkunuri, 1989). However, software development methods of the past have encouraged the creation of long modules, full of functionality, which are highly connected and dependent on each other (Ince, 1991). The procedure-oriented (PO) paradigm views programs as collections of interacting functions and procedures (Purchase and Winder, 1991). The primary conceptual units of programming are the procedures, which usually take the form of lengthy functions or routines with long series of arguments, and the principal structuring mechanism is the nesting of procedure invocations. The code is therefore decomposed according to an *algorithmic approach*. These procedures can seldom be re-used without modification, which can range from changing an argument list to re-writing the program from scratch. The data variables and values are passive, and are either global to all the procedures, local to one, or passed between them as necessary (Crenshaw, 1991).

Object-oriented programming concepts, which aim to significantly increase modularity and reusability, were first introduced during the 1960's, with the programming language Simula (Dahl and Nygaard, 1966). Since the days of Simula, the OOP approach has been implemented in a variety of languages and environments. OOP langages can be divided into two groups: pure and hybrid (Yip and Dessey, 1991). Pure languages such as Smalltalk (Goldberg and Robson, 1983; Goldberg, 1984) and Eiffel (Meyer, 1988) are designed from the beginning according to OOP principles. Hybrid languages such as C++ (Stroustrup, 1986), Objective C (Cox, 1986) and CLOS (Keene, 1989) are extensions of existing non-object oriented languages, so that the resulting language has the ability to implement fundamental object-oriented concepts. This approach allows programmers to learn the new concepts in a more gradual way and with less frustration. However, a stronger programming discipline is required in this case, since it is easy for programmers unfamiliar with the new principles to slide back into a procedural programming style.

Of all the pure OOP languages, Smalltalk is perhaps the most commonly referred to, as it spearheaded the object-oriented era in computing. Since, in the early 70's, Alan Kay and others developed the Smalltalk system at the Software Concepts Group of the Xerox Palo Alto Research Center, Smalltalk has undergone a long series of modifications and improvements (Wirfs-Brock, 1991). In 1983, Xerox released the first commercially available version of Smalltalk-80; a class library and virtual machine for the IBM PC (called "Methods") was released by Digitalk in the same year. A successor product, called Smalltalk/V, followed in 1986. In 1987, Adele Goldberg and others formed a spinout company called ParcPlace Systems, which is the one today associated with current versions of Smalltalk. The most significant change Smalltalk has undergone is perhaps the fact that the latest versions run under control of the host computer operating system, rather than being standalone systems like the experimental versions at Xerox. Furthemore, the latest version (Objectworks/Smalltalk 4.0 from ParcPlace Systems) now has the ability to use the standard window manager of the host computer instead of taking total control of the display environment. As an object-oriented programming pioneer, Smalltalk had to develop its own graphics kernel, window system, and multitasking executive. Nowadays, it has had to delegate to its supporting platform a number of the functions it provided, in order to guarantee compatibility with the graphical user interfaces it inspired (Udell, 1990).

Smalltalk's advantage over its competitors is the sophistication of its environment and the cleanliness of its implementation of the object-oriented paradigm. It comprises the Smalltalk language, extensive class libraries, and the programming environment that enables a programmer to input, test and run Smalltalk applications. Among the facilities provided by the environment are a set of tools consisting of a graphics interface, code and object browsers, symbolic debuggers, and many other facilities besides the language compiler itself (Duimovich and Milinkovich, 1991). However, the insularity of the environment is one of Smalltalk's disadvantages normally pointed out, as normal Smalltalk applications cannot be separated from the environment and must be run from within it.

As an alternative, C++ implements all the major object-oriented features and is one of the most widely used object-oriented languages (Ege, 1992). The history of C++ is not as exciting as Smalltalk's, mainly because it was born in an era when object-oriented programming had already been pioneered by Smalltalk and its benefits were becoming apparent. The ancestor of C++, C-with-classes, was born in 1980 (Coplien, 1992). By the summer of 1983, C-with-classes had entered the academic and research world. Further developments, which led to C++, benefited from the experience of users and from the progress in object-oriented programming — still a maturing paradigm — in the rest of the computing community. In C++, a balance had to be made between the influence of C (Kernighan and Ritchie, 1978), which provides efficiency and close access to the machine, and the implementation of the new concepts. An overview of the main differences between C and C++ is made by Jordan (1990).

Plain C++ does not offer standard library classes like Smalltalk, nor does it come with a programming environment. External libraries must be developed or bought for more advanced applications, such as graphical user interfaces.

The National Institute of Health, in the United States, created the best known class library for C++ (Gorlen *et al.*, 1990), which is loosely modelled after the Smalltalk class library (Ege, 1992). Another library of classes has been developed by the Free Software Foundation, the GNU C++ class library. Although intensive use of developed class libraries is the basis of reusability, compatibility of classes from different libraries is still not guaranteed (Ege, 1992).

A recent software tool (ObjectVision, 1990) allows the visual programming of objects. It is possible to build object-oriented applications simply by drawing the desired hierarchy of objects on the screen; ObjectVision then converts the diagram into commented, ready to compile C++ or TurboPascal code. Further details are given in the ObjectVision Package and Manual document (ObjectVision, 1990). A PC version of C++, Turbo C++ (Borland, 1990) has also recently come up with the typical Borland environment, which includes an editor and a debugger.

Other languages which are accepted or gaining wide acceptance are Objective-C, Eiffel, an object-oriented version of Pascal (i.e., Turbo-Pascal) and the Common Lisp Object System (CLOS) (Keene, 1989).

Objective-C is another hybrid language which, like C++, extends the programming language, but in a Smalltalk style which shows in its terms and basic concepts. It is a mixture of C and Smalltalk; for example, while it uses Smalltalk message sending style, it reverts to C for its control structures. It comes with a large standard library class that allows the construction of new applications from pre-defined components. The classes are organized into two main libraries: the basic class library ICPak101 and the graphics class library ICPak201, which provides classes to allow the development of graphical user interfaces. ICPak201 is an user interface toolkit written in Objective C (Knolle, 1989). Furthermore, Objective C also provides a compiler, class browser and tracing and debugging facilities in the programming environment.

Eiffel (Meyer, 1988) is a newer entry in the set of object-oriented programming languages. It is a pure language in the sense that it was developed from scratch to fully support all the object-oriented concepts. The basic style of Eiffel has a Pascal flavour (Ege, 1992). It provides a large library of pre-defined classes which range from kernel and data structure classes to advanced graphic classes. It also provides several useful tools for program development such as a compiler, class browser, and an editor.

Hybrid object-oriented versions of Pascal are the latest appearances in the OOP language market. Turbo Pascal from Borland International and Quick Pascal from Microsoft have extended Pascal with most of the object-oriented concepts. For example, TurboPascal supports the basic concepts of object, class, method, message, and inheritance. The contribution given by object-oriented versions of Pascal is again that, like extensions of C, they open a learning path for many Pascal trained programmers.

Finally, the Common Lisp Object System was designed, starting in 1986, as an attempt to unify previous versions of object-oriented extensions to Common Lisp (Keene, 1989). One of them was Flavors, developed by Symbolics Inc. (Symbolics Inc, 1986), and the other was CommonLoops, developed by Xerox PARC.

A detailed comparative survey of object-oriented programming languages is made by Micallef (1988) and an overview of other developments, such as object-oriented COBOL, Trellis, BETA, and others, is presented by Graham (1994).

2.4.1 Brief Comparison Between Smalltalk and C++

Smalltalk has a complete object-oriented organization throughout and does not allow stand-alone classes. This means that it is not possible to define a class which does not have a superclass. The root of Smalltalk class hierarchy is class "Object", which implies that all the classes are subclasses of this class. Smalltalk is a weakly typed language; its variables can refer to an object of any class. In such languages, little emphasis is placed on defining the type of objects at compile time, and it is often unknown until run-time. This is in practice done by allowing the programmer to specify code without an immediate check of whether or not it will execute correctly. Since methods are looked up at run time, one of Smalltalk's typical errors is that a certain message cannot be understood.

C++ is a strongly typed language, which means that the exact type of each and every object is explicitly stated in the source code. Although more work has to be performed by the compiler to thoroughly type-check all the program, strongly-typed languages tend to produce faster and more robust executables. The source code is however less flexible since any modification in the types used will lead to re-writing a portion of the program. As the advantages of both weak and strong typing are still a subject of discussion, it is likely that both will continue to co-exist.

In Smalltalk it is not possible to hide an instance method. Encapsulation is achieved by hiding all instance variables and exposing all instance methods. C++, in contrast, supports a powerful series of different encapsulation measures (private, protected, friend, public). Its capabilities to control access to the members of a class are the best of all the object-oriented languages.

Smalltalk and C++ support inheritance like all object-oriented programming languages. The new subclasses may well have their own version of existing member functions. In C++, to ensure that the method belonging to the new class will be called, rather than the method that belongs to its superclass, the superclass must have that method declared as virtual if it is likely that it may be extended in that way. Functions are always virtual in some OOP languages for this reason. However, in C++ it is up to the programmer to specify which functions are virtual. The disadvantage of declaring a function virtual is that the program has to determine which function to call at run-time, which is more time-consuming (Mitchell, 1993).

Smalltalk does not support multiple inheritance, that is, the capacity of a class to inherit from more than one parent classes, as provided by C++. On some occasions this restriction is undesirable and leads to unnatural coding styles or the duplication of code. Although earlier versions of Smalltalk-80 allowed multiple inheritance (Borning and Ingalls, 1982), this feature was never well supported and finally dropped in the latest version of Objectworks/Smalltalk-80.

2.5 Exception Handling Mechanisms

Exception handling mechanisms are a means for a part of the program to inform another part (which can hopefully do something sensible about it) that an "exceptional circumstance" has been detected.

Object-oriented languages can be classified into three categories, depending on their exception-handling possibilities (Dony, 1988). In languages such as Smalltalk, handlers for all exceptions can only be attached statically to classes. Languages which are extensions of procedural languages have in general been done without modification of the standard or existing exception handling mechanisms (e.g. C++) and do not provide solutions to associate handlers with classes. Finally, other languages built on top of procedural or functional ones (e.g. Loops or Objvlisp) choose a compromise. Some exceptions relative to object manipulation are raised and handled as in Smalltalk, whereas the other ones depend on the underlying language. It is impossible, for example, to attach a handler for *divide-by-zero* to a class in this last category. As a consequence, the handling possibilities are unequal. A discussion and comparison of several existing exception-handling strategies is made by Dony (1988) and Dony *el al.* (1992); Dony (1988, 1990) proposes a specification of an object-oriented exception handling system.

Koenig and Stroustrup (1990) have proposed a mechanism for exception handling in C++. The process proposed consists of raising objects and catching classes; a copy of the object raised is passed to the handler. For example, in the case of an integer overflow happening, an object of the class Int_overflow would be created by the function where the exception happened. A copy of this object would be caught by the exception handler, which in turn is declared to accept objects of class Int_overflow. The type of object raised is used to select the handler in approximately the way in which the arguments of a function call are used to select an overloaded function. The classification of exceptions into classes is a natural one; for example, one could imagine a Matherr exception that includes Overflow, Underflow and other possible exceptions. The syntax of the handling mechanisms is described by Koenig and Stroustrup (1990) and Coplien (1992) and includes new keywords like *try* and *catch*. However, these mechanisms are not part of C++ yet. Most C++ environments, such as the C++ compiler used in this work, do not support exception handling other than the one already provided by C.

In conclusion, the object-oriented programming paradigm offers several advantages over the traditional procedure-oriented paradigm, namely increased modularity and re-usability. A large number of object-oriented languages has been developed in recent years; Smalltalk and C++ are among the most successful. Object-oriented concepts are invaluable for the development of graphical user interfaces, as will be shown in the next chapter.

Chapter 3

Graphical User Interfaces

3.1 Introduction

Software that functions but is difficult or confusing to use rarely actually gets to do anything. (Rimmer, 1992)

During the last decade, the computer world has been revolutionized by a steep decrease in hardware price and by the development of increasingly powerful computers, which has enabled the use of windowing software. Computers are now used by a very broad range of users, and for a large variety of tasks. As a consequence, the importance of user interfaces has risen accordingly. Based on software ergonomic investigations, the conclusion that graphical user interfaces (GUI's) are the interactive man/machine means of communication of the future is now widely accepted (Encarnação *et al.*, 1991). Recent studies have shown that users in a GUI environment work faster, more accurately and with lower frustration and fatigue levels than users in a text-based environment. Considerable increases in both productivity and accuracy of completed work are achieved using GUI's (Peddie, 1992). As audio/speech technology becomes increasingly available, research is currently being conducted on the conjunction of audio techniques and visual ones, which seems to result in a significant improvement in performance (Williams *et al.*, 1990).

However, Human-Computer Interaction (HCI) is a discipline that integrates both psychology and computer science (Harrison and Thimbleby, 1990). The fusion of these two subjects is not simple and research work on this area often takes an approach which is specific to one or the other. Whereas the computing approach usually focusses on the development of toolkits, frameworks, and windowing and graphics systems, the psychological approach is almost exclusively devoted to what is known as human factors, including cognitive aspects, user models and evaluation. An overview of both approaches is made in this chapter.

3.2 Type of Interaction

In batch systems, the interaction between the system and the user is restricted to the submission of the batch job. The first generation of user interfaces was therefore not interactive (Nielsen, 1993), since all the user's commands had to be specified as a whole before the result was known. The next generation of interfaces consisted of text-based, line-oriented interfaces (Nielsen, 1993). A single line acted as the command line and, after the <Return> key had been hit, no further changes could be made to the command. Initially, these interfaces were implemented on tele-typewriters (TTY's), and later in terminal screens. Because the text was "frozen" as soon as it scrolled above the command line, these interfaces were sometimes called "glass TTY's". Some time later, full-screen interfaces were introduced, taking advantage of the modifiable nature of the entire screen. An example of such an interface, still widely used today, is vi (visual editor) on UNIX systems. Performance times with full-screen display editors are about half as long as with line-oriented editors (Shneiderman, 1992), which represents a considerable improvement.

As early as 1962, Ivan Sutherland's Sketchpad system (Sutherland, 1963), a drawing tool based on the use of a light pen, was developed, and is today often considered the first graphical user interface. However, GUI's did not see widespread use until the early 1980's. The graphical user interface for the Star computer (Smith *et al.*, 1982), developed in 1982 at Xerox Corporation, Palo Alto, USA, introduced concepts which are still used by most GUI's. It presented the use of the Physical Office Metaphor, today commonly known as the Desktop Metaphor, in which physical objects in an office such as paper, folders, file cabinets, mail boxes, etc. are represented by icons on the display screen. Selection was made by pointing to the desired item of data with the mouse and clicking the mouse buttons, thus introducing a practical example of *direct-manipulation*, as will be discussed later. The Star also introduced new commands, "universal" or generic commands (MOVE, COPY, DELETE, SHOW PROPERTIES, etc.), that could be used throughout the system. Each of them performed the operation as suited to the type of object currently selected, as is still done today by the Macintosh GUI. This probably represented one of the earliest uses of the object-oriented approach in graphical user interfaces.

Shneiderman (1983) was the first to use the term *direct-manipulation.* Since then, both the terminology and the interaction style have become increasingly popular. Direct-manipulation interfaces are normally associated with WIMP (Windows, Icons, Menus and Pointing Devices) interfaces. Note that keyboards remain the most efficient tool for the entry of text and digits (Deininger and Fernandez, 1990). Exactly what defines the direct-manipulation type of behaviour is often unclear (Harrison and Dix, 1990). Main characteristics are that there is a close relationship between input and function (often one keystroke or mouse click per command) and that the data manipulated by the user interface (and therefore the effects of all commands) are immediately visible; the user is provided with a response in real time that represents the action taken (Deininger and Fernandez, 1990). This gives the user the impression of operating directly on objects in the computer, rather than carrying on a dialogue about them (Jacob, 1986). Moreover, the designer is forced to find simple solutions for the problems posed by the interface, since if something is hidden it cannot be used, thus avoiding hidden and sometimes unnecessary complexity (Thimbleby, 1990). In a direct-manipulation interface, the entry of commands in some (usually) arbitrary language is replaced by a "point and shoot" style of interaction, where the fundamental operation is a selection rather than a command (Chignell and Waterworth, 1991).

Soon after the Star, WIMP interfaces spread to the Macintosh computer via the graphical user interface for the Lisa computer (Williams, 1983). Later, they spread from the Macintosh to other types of microcomputers and, with the development of windowing systems such as X Window, to UNIX workstations.

WIMP interfaces provide several advantages, allowing the user to visualize the progress of the task, recognize operations instead of having to recall commands, transfer knowledge about the architecture of the physical interface from one application to another and achieve a direct relationship between input and response (Chignell and Waterworth, 1991). Newer interaction styles, such as virtual reality and remote control, are further topics for research (Shneiderman, 1992).

3.2.1 Mode

Mode can be defined as "the variable information in the computer system affecting the meaning of what the user sees and does" (Thimbleby, 1990), or "a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input" (Larry Tesler, cited in Smith *et al.*, 1982). A user interface may give different meanings to the same input, depending on the current mode. Modes are confusing and a frequent source of user error and frustration (Nielsen, 1993), especially because a computer system usually gives the user no clue as to how it got to its present state or what exactly that state is (Thimbleby, 1990). If they cannot be completely avoided, they should be explicitly recognized in the interface design. If states are shown clearly to the user, feedback can be provided, thus helping the user not to mistake the current mode. As Jacob (1986) points out, although graphical user interfaces appear to be modeless, in reality they have many distinct modes. Direct-manipulation user interfaces divide the screen into small mode areas — areas on the screen that interact with the user differently from the surrounding areas. Different modes are accessed moving the cursor to a different object on the screen, because the range of acceptable inputs is reduced and the meaning of each of these is determined. However, the disadvantages associated with moded interfaces do not appear to be present in direct-manipulation interfaces. An explanation to this lies on the fact that mode is so obvious and easy to change that it no longer causes confusion or effort to change (Jacob, 1986).

3.3 Human Factors Considerations

Although "design is a creative process and hence there are few absolutes that must be adhered to" (Rubin, 1988) human factors considerations must be incorporated in every stage of the design of user interfaces. Failure to do so results in tiring, confusing or awkward to use systems. This is critical in applications like medical systems, air traffic control or nuclear power plant control, where it can result in the loss of lives or a major disaster. The number of recommended guidelines has risen exponentially during the last few years. Typical collections sometimes reach more than a thousand rules to follow (see Apple, 1987; Gilmore *et al.*, 1989; Rubin, 1988; Shneiderman, 1992; Smith and Mosier, 1986) and, if anything, the sheer number of guidelines available is more of a hindrance than a help. Furthermore, some of them are contradictory in their design recommendations. In this section, a brief overview of the main areas normally covered by human factors is made.

3.3.1 Consistency

Although the exact definition of *consistency* is still unclear, it has become a very important feature in user interface design (Schiele and Green, 1990). It refers to regularity in various aspects of the interface, which allows users to make generalizations based on their previous knowledge. If users know that the same action has always the same effect, they feel more confident to use the system, and are encouraged to explore it, thus improving the learning process (Nielsen, 1993). Several aspects must be considered, namely the actions necessary to perform tasks, the feedback provided by the system, the spatial layout of the screen, the appearance of visual objects, etc.. An example of inconsistency is, in Unix workstations, the fact that the "delete" operation may be invoked in a variety of ways depending on context: deleting a window by pointing to it with a mouse and selecting an operation from a pop-up menu; deleting a file by typing "rm" and the file name; deleting a character from a file by invoking an editor, moving the cursor to the desired location, and typing 'x', etc. (Dewan and Solomon, 1990). In spite of the importance given to consistency, Pangalos (1992b) states that few users use completely consistent user interfaces.

TAG (Task Action Grammar) (Schiele and Green, 1990) has been proposed to measure the property of consistency formally and quantitatively. When several tasks are similar, and are achieved by actions which are similar, the how-to-do it knowledge can be expressed by a schema which encapsulates several low-level, individual rules. The number of schemas that describe an interface gives an indication of the consistency or quirkiness of the interface language; the proponents of TAG believe that the fewer the schemas, the more likely it is that the user can generalise from partial knowledge. TAG therefore represents an attempt to use a formal notation to represent a human factors concept.

3.3.2 Use of Colour

Although colour is an excellent means of improving the comprehensibility of displays such as pie and bar charts (Rubin, 1988), the use of colour must be judicious to avoid visual interaction that results in communication-damaging noise and that can produce strong after-effects. According to cartographic principles, background dull colours are more effective, allowing the smaller, bright areas to stand out with greater vividness (Tufte, 1992). Rubin (1988) also states that the use of colour in a man-machine interface should be as consistent as possible with everyday usage.

3.3.3 Minimization of User Memory Load

The limitations in the way the human brain processes information requires that displays be kept simple. Every additional feature or item of information on a screen is something else to learn, possibly misunderstand, and search through. Displaying too many objects and attributes results in a relative loss of salience for the ones of interest to the user (Nielsen, 1993).

The terminology used should be based on the user's language and not on system-oriented terms. Commands should be consistent, "guessable" and, wherever possible, described by short words. Where appropriate, on-line access to command-syntax forms, abbreviations, codes, and other information should be provided (Shneiderman, 1992).

To minimize the users' memory load, the system should be based on a small number of pervasive rules that apply throughout the interface. If a very large number of rules applies, the user has to learn and remember all of them, which is a burden (Nielsen, 1993). For more on human short and long term memory, see Thimbleby (1990).

3.3.4 Feedback

It is in general far easier for a user to recognize information than having to recall it without help (Nielsen, 1993). As an example, when learning a foreign language, the passive vocabulary is always much larger than the active vocabulary. The system should keep the user updated about what it is doing and how it is interpreting the user's input, thus providing positive feedback (even if only partial) as soon as it becomes available. Feedback should not wait until an error situation has occurred; for every action, there should be some system feedback (Shneiderman, 1992), although the response can be modest for frequent and minor actions.

3.3.5 Response Time

Response time is closely related to the issue of feedback. If the system has long response times for certain operations, feedback becomes especially important. For delays longer than 10 seconds, users will want to perform other tasks while waiting for the computer to finish. Feedback is thus necessary as to when the computer expects to be done. Rubin (1988) presents a table of recommended system response times, which range from 0.1 seconds (for the effect of the movement of an input device such as a mouse to be detected) up to 5 seconds (request for a complex service).

3.3.6 Easy Escape Routes

To avoid giving the user the feeling of being trapped by the computer, thus increasing his/her feeling of control over the dialogue, ways out should be offered of most situations. For example, all dialog boxes and system states should have a clearly marked exit, such as a cancel button or other escape facility (Nielsen, 1993).

3.3.7 Adequacy to User's Level of Expertise

The experienced user usually wants to perform frequent operations especially fast, using dialogue shortcuts. Typical accelerators include abbreviations, special keys that package an entire command in a single keypress, clicking on an object to perform the most common operation on it, buttons available to access important functions directly, and the possibility of defining macro commands (Nielsen, 1993; Shneiderman, 1992).

3.3.8 Error Messages

Error situations represent situations where the user is in trouble and is potentially unable to use the system to achieve the desired goal. They also present opportunities to help the user understand the system better, since at this stage the user is normally alert and will pay attention to the contents of error messages (Nielsen, 1993). However, better than having good error messages is to avoid the error situation; as much as possible, the system should be designed so that the user cannot make a serious error (Shneiderman, 1992). Erroneous commands should leave the system unchanged, or it should give instructions about restoring its state. Many situations are error-prone and systems can often be designed to avoid them, for example by asking the user to select a filename from a menu rather than typing it. If a user is about to save a document with the same name as an existing one, he should be warned before the action is completed. "Undo" facilities are also useful. The reversibility of actions relieves anxiety since the user knows that errors can be undone, thus encouraging the exploration of unfamiliar options (Shneiderman, 1992). If an error still occurs, error messages should be clear and concise (Rubin, 1988). They should be polite and never imply blame on the part of the user; furthemore, they should offer some constructive advice, to help the user recover from the error.

3.3.9 Help and Documentation

Besides the fact that there is no need to learn the command language, which allows for productivity in a very short time, menu-based interfaces offer several other advantages (Kantorowitz and Sudarsky, 1989): the possibility of exploring the operations provided by simply browsing through the menus and, if an adequate help system is installed, no need for a manual in most cases. However, in some cases, documentation cannot be avoided. Most user interfaces have sufficiently many features to warrant a manual and possibly an off-line help system. Also, regular users of a system may want documentation to enable them to acquire higher levels of expertise, which should be available. Rubin (1988) states that *any* kind of help can bring a dramatic improvement in performance, and that allowing users to choose when to initiate a help request is better in terms of speed of task and frequency of errors. Lastly, at least novice users seem to prefer hard-copy help rather than on-screen help.

3.3.10 The User-centred Approach

As the number and variety of users increased, user interface designers recognized the need to learn more about the user (Marcus and Van Dam, 1991). An example of this user-centred approach is the capability that lets users alter the look and feel of their user interfaces, selecting characteristics such as colour and layout. In the future, it may mean users will be able to decide on feedback modes, for example audio or visual.

Most people do not acquire expertise in all parts of a system, in spite of the fequency of usage (Nielsen, 1993). The degree of expertise of a user can be, moreover, classified according to several different sets of dimensions. Nielsen (1993) proposes that users' experience be considered to differ along three main dimensions: experience with the system, with computers in general, and with the task domain itself (see Fig.3.1). Other possible dimensions are the degree to which the user was the producer or consumer of information, whether the user had any part in developing the system, and the authority the user possessed concerning decisions over the system (Nielsen, 1993).

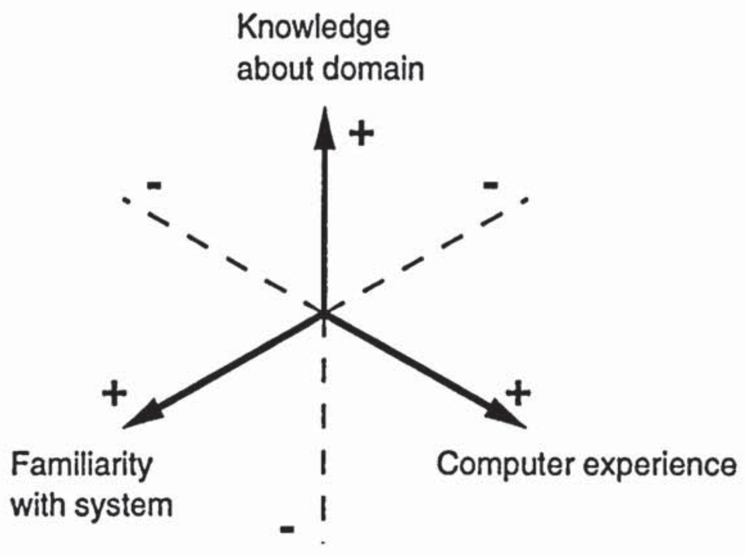


Figure 3.1: Three main dimensions along which users' experience differs. (Adapted from Nielsen, 1993).

In practice, differences between user levels of expertise depend on the quality of the model which the user has of the system. Rubin (1988) distinguishes between three basic models: the Design Model (DM), the User's Model (UM) and the System Image (SI) (see Fig.3.2). The Design Model is the conceptual model that the designer has as a whole of the system to be built. The User's Model is the mental model that the user holds of the system. It is each individual's personalized interpretation of the system. The System Image is what everybody that interacts with the system sees, hears and feels; it is therefore what the system offers and presents, in objective terms, to the human users, as well as the accompanying literature and training material. The designer aims to produce a system and a SI that, as accurately as possible, corresponds to the nature and working of the conceptual model (DM). As an example, the designer may have designed a drawing tool for three dimensional sketching. However, if the rotation operation can only be accessed after a long search through several menu levels, whereas the move and resize operations are displayed in the menu bar, the user may not realize it is possible to rotate graphical elements at all. In this case, the User's Model is not close to the Design Model, mainly because the System Image is not ideal in the sense that it does not offer the user a clear idea

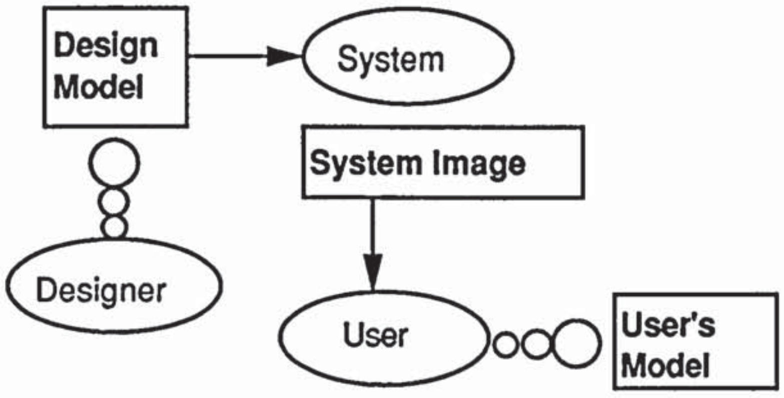


Figure 3.2: The Design Model, System Image and User's Model. (Adapted from Rubin, 1988).

of all the available operations. The better the SI reflects the designer's intentions (the DM) the closer the model formed by the user (the UM) will be to the DM. If the UM is close to the DM then the user will be able to exploit the design to its full potential, since the model that a user has of an interactive system governs his/her interactions with it (Barfield, 1993). Careful consideration must therefore be taken during design and implementation so that the mental model the user develops is clear and close to the real system. Note that the existence of consistency is an important factor for the construction of a coherent user model.

3.4 User Interface Analysis and Design

The user interface is not the only component of an interactive software system; the functionality of the application for which the interface is being developed must also be considered. Hix and Hartson (1993) recommend that, for interactive software systems, analysis be decomposed into several stages:

The *needs analysis* establishes that a new system is in fact needed, and determines the basic goals, purpose, and features desired.

The *user analysis* combines specific information about job functions and tasks of potential users, plus social and organizational considerations. The result is a set of user class definitions.

The *task analysis* — defined by Sutcliffe and McDermott (1991) as the HCI equivalent to systems analysis — provides a complete description of the tasks, subtasks, and methods involved in using the new system, identifying resources necessary for the users and the system to perform these tasks cooperatively. Task analysis usually results in a top-down decomposition of detailed task descriptions. The *functional analysis*, similar to task analysis, results in an internal view of the technical functions to be designed into the computational (non-interface) component of the system.

The *task/function allocation* produces decisions about which parts of the tasks will be performed by the human user and which will be performed by the system. Some tasks may be manual (user) while others are automated (system).

The *requirements analysis*, as mentioned in Chapter 2, is the formal process of specifying design requirements for the system. In order to set formal requirements for design, requirements analysis is based on needs analysis, user analysis, task analysis, and functional analysis.

In recent years, several methods for analysis, specification and design of the user interface have been proposed in order to integrate the development of the interface with structured systems analysis and design. Another aim is to cover as much as possible of the systems design life cycle. Kuo and Karimi (1988) have proposed the use of Data Flow Diagrams (DFD) to derive user interface specifications in a systematic way. Sutcliffe and McDermott (1991) have proposed a five-step method which covers task and user analysis, interface specification and dialogue design. Cugini (1989) also proposes a methodology to address the problem of user interface design, based on a top-down approach, in which the analysis scheme is based on several different levels.

Most of these methods involve the use, at some point, of formal notations.

3.4.1 Formal Methods

Formal specification techniques are useful because they can specify user interface behaviour, independently of software implementation (Kuo and Karimi, 1988). In addition, user interface specifications can be used to predict user errors before implementation.

The most frequent techniques are those which draw on the linguistic model and are derived from traditional language specification and analysis, namely

BNF (Backus-Naur form) and state-transition diagrams (Alexander, 1987; Green, 1985; Jacob, 1983). Frame-based and knowledge-based techniques are also referenced by Alexander (1987). The User Action Notation (UAN) is a new notation developed specifically for the specification of user interfaces (Hix and Hartson, 1993). Most interfaces today include both sequential and asynchronous styles of interaction. In an asynchronous style, many tasks are available to a user at one time, and the state of each task is independent of the state of the other tasks. Other notations adequate for asynchronous interaction are based on events (Green, 1985) and object-oriented techniques (Alexander,1987; Jacob, 1986). These last two are relevant to this work and are introduced next.

The notation for the dialogue control component based on events or event handlers, loosely based on the object-oriented approach to user interface design, is especially adequate for asynchronous interaction (Green, 1985). Each user input or system output is viewed as an event and is sent to the appropriate event handler, a software routine associated with a class of events. When an event handler receives an event the associated procedure is executed. These procedures can perform calculations, send events to other event handlers, and send tokens to the presentation component and application interface model. The dialogue control component consists of a collection of event handlers that can change dynamically.

Another technique uses an object-oriented approach (Alexander, 1987; Jacob, 1986). Note that an object-oriented approach does not necessarily imply an object-oriented language, although the implementation of the underlying concepts are significantly easier if one is used. Jacob (1986) defines the interface using *interaction objects.* Each locus of dialogue is described as a separate object with a single-thread state diagram, which can be suspended and resumed, but retains state. In the object-oriented style, each object specifies its data and procedures, as well as its dialogue with the user. High-level object classes contain default behaviour, yet can be specialized for specific behaviour. This inheritance mechanism is provided to avoid repetitiveness in the specifications.

There are some important similarities and differences between the event-based and object-oriented techniques. The event handlers perform the same function as objects and classes — receive events, or messages, and take appropriate action. However, there is no explicit inheritance mechanism for event handlers. Furthermore, as Green (1985) points out, messages in OOP are synchronous whereas events are asynchronous. When an object sends a message, it suspends its execution and transfers control to the receiving object. When the receiving object completes its computation, control returns to the sending object. In the case of events there is no handshaking betwen the sending and receiving event handlers. The sending event handler may not suspend its execution when it generates an event, and the receiving event handler may receive the event any time after it was generated.

With the development of windowing systems such as X Window, which will be described in greater detail in the next section and in Chapter 4, the structure of the interface is best described by a fusion of event handling and object-oriented techniques. Event handlers are associated with objects, and therefore the distinction between *event handlers* and *methods* is not clear.

3.5 Toolkits and Graphics Systems

Although standardization has many benefits, at present there is a considerable number of graphics and windowing systems. Examples of windowing systems and graphical user interfaces are the X Window System for UNIX workstations, MS Windows and Presentation Manager for IBM Personal Computers, the Macintosh software for Apple computers, GEM for the Atari and Intuition for the Amiga (Nicholls, 1990; Pangalos, 1992a). A characteristic that varies widely is the level of integration between the GUI and the operating system. Some GUI's are tightly bound to the system, appearing automatically when the computer is turned on (Hayes and Baran, 1989), such as the Mac, Amiga, or NeXT computers. By contrast, Microsoft Windows and most of the X Window GUI's that run under Unix must be specifically invoked.

In the UNIX-based world the *de facto* standard, developed by MIT (Scheifler and Gettys, 1986), is usually known as the X Window System or simply X. It is not a GUI, but rather a common windowing system across networks connecting machines from different vendors, as well as a foundation on which to build graphical user interfaces (Peddie, 1992). Since the X Window system is in the public domain and not particular to any hardware platform or operating system, and its distribution is not subject to the commercial interests of a single supplier, it may well become the standard windowing environment for all sorts of computers. Any machine that supports multitasking and interprocess communication and has a C compiler can support X (Moore, 1990). The lowest X programming level is the X Protocol. It is similar to machine language, and programming with it is complex, cumbersome and error-prone. On the next higher level, Xlib is a library of over 300 C language functions used to generate X Protocol. In practical terms, X relies on the development of graphical user interface toolkits, which make extensive use of Xlib function calls. Toolkits are sets of widgets, or building blocks, such as dialogue boxes, menus, buttons, scroll bars, etc. — each with a consistent appearance and behaviour, that assist the application developer do his work. They are usually implemented in the form of libraries of user interface objects. X Toolkits employ concepts used in object-oriented programming; this is quite a feat as they are written in and for the C language, which does not directly support objects (Miller, 1990). The X Toolkit Intrinsics, also known as Xt Intrinsics, is part of the X Window system itself (Lainhart, 1991). The uppermost programming level for X Window, and the one currently used by most programmers, consists of extensions to the X Toolkit Intrinsics and is proprietary. Proprietary toolkits come with their own window manager — the window manager is a special application which is responsible for manipulating windows on the screen — and graphical user interface. A windowing system does not, *per se*, provide a direct-manipulation environment, since the user still has to type a command line for file management or to start any program (Judge, 1990). The "desktop manager" is an application program, which uses icons to give access to all the other programs, and turns file management, communications and administration into visual tasks, guiding the novice user through cryptic UNIX commands using dialogue boxes. Both Motif (developed by Open Software Foundation and a number of others such as DEC, Hewlett-Packard, Microsoft, etc.) and OpenLook (AT&T and Sun) style interfaces are implemented on top of X by proprietary toolkits which provide a consistent object look and feel throughout the interface. A comparison between the OpenLook and the Motif toolkits need not be made here. Advantages of one over the other seem to rely on portability alone, which in turn relies on the number of vendors and third-party supporters. No other major differences have been reported. One of the current arguments is over which X-based GUI, especially Motif or OpenLook, should be the standard. It is not a trivial question and not something to be dismissed lightly, as there is a large amount of money involved. However, this is a commercial problem rather than a technical one.

Applications written for X at the lowest levels are synchronous and procedural. Somewhere in the heart of the program there is an infinite loop and a switch/case statement, with appropriate procedure calls made to respond to each incoming event. With an application containing many windows, this can be a difficult program to understand and maintain. The toolkits, mercifully, hide this complexity from the programmer. They present a different and more useful model of event-driven programming, which is apparently asynchronous and object-oriented (Lainhart, 1991).

A library of user interface objects enables the programmer to write the GUI code in terms of high-level function calls, thus hiding much of the complexity of GUI programming. However, since even relatively simple applications require hundreds of lines of code to run in a graphical environment, the use of traditional languages like C still imposes a steep learning curve and long development times (Urlocker, 1989). A survey of commercial programs showed that display generation and management code constituted 40–60% of the source text of the programs sampled (Dewan and Solomon, 1990). The demand placed on programmers to create structured, modular, portable, and reusable code, and the complexity of developing code for GUI's, have made object-oriented programming (OOP) the development methodology of choice. Objects are natural for representing the elements of a user interface and supporting their direct manipulation. Experience shows that user interfaces written in object-oriented languages are significantly easier to develop and maintain (Linton *et al.*, 1989). Unfortunately, experience also shows that developers who are used to designing function-oriented user interfaces have serious difficulties in changing over to designing object-oriented interfaces (Nielsen, 1993).

There are relatively few applications that run under X Window today. It may be some years before applications are generated at a higher rate, much the same situation that was experienced by Microsoft Windows. It will be just a matter of time, however, until there is a wide selection to choose from (Peddie, 1992). Since X Window was the windowing system used in this work, together with the X Window-based toolkit XView, a brief description of both follows.

3.5.1 The X Window System

X Window, or simply X, is the result of two separate groups at MIT (USA) having a simultaneous and urgent need for a windowing system (Scheifler and Gettys, 1986). In the summer of 1984, the Argus System at the Laboratory for Computer Science needed a debugging environment for multiprocess distributed applications; Project Athena was faced with large numbers of workstations with bitmap displays and needed a windowing system to make the displays useful. An operating system-independent windowing system seemed to be the only solution for both problems, leading to the development of X.

X differs from all previous systems in the degree to which both graphics functions and "system" functions are application functions, and in the ability to tailor desktop management transparently. X can display output windows from several applications at once in one screen, and these programs can run simultaneously on different machines. The machines do not have to be the same type or run the same operating system (Moore, 1990).

One of the goals of X was to impose no style of interface on its users, and to provide *hooks* (mechanism) rather than *religion* (policy). For example, since menu styles vary dramatically among different user interfaces, the window system must provide primitives from which menus can be built, instead of just providing a fixed menu facility (Scheifler and Gettys, 1986).

The base window system is the substrate on which applications, including special applications like the window and input managers, are built. The window manager implements the desktop portion of the management, controlling the size and placement of application windows, and sometimes application window attributes, such as titles and borders. The input manager implements the remainder of the management interface, controlling which applications see input from which devices (e.g., keyboard and mouse).

The basic characteristics of the X base window system are listed next (greater detail is given by Scheifler and Gettys, 1986). The system

— is implementable on a variety of displays;

— is network-transparent;

— supports multiple applications concurrently;

— can support many different management interfaces;

— supports overlapping windows, including output to partially obscured windows;

— supports a hierarchy of resizable windows, and an application is able to use many windows at once;

— provides high-performance, high-quality support for text, 2-D graphics, and imaging;

— is extensible.

Finally, applications are device-independent.

The client/server architecture of X Window is presented in greater detail in Chapter 4.

3.5.2 The XView Toolkit

Although written in the C language, XView behaves as an object-oriented toolkit. All the widgets can be considered objects from specific packages, a set of properties being associated with each package. XView is based on the fundamental principles of object-oriented programming (Heller, 1993):

— objects are represented in a class hierarchy;

— objects have attributes which can be set via message passing functions.

Furthermore, objects may have associated procedures that are triggered by events (event handlers or callback procedures), which correspond to *methods* in OOP.

For example, frame is a subclass of the class window, which is in turn a subclass of the class drawable. In XView, a package is often referred to as a class,

Figure 3.3: XView class hierarchy.
(From Heller, 1993).

meaning a set of related functional elements. The classing system is extensible, so new classes, based or not on existing ones, can be created. Chain inheritance is used as a part of XView's object-oriented model. This means that all objects in a particular class inherit the properties of the parent class (or super class). Each class contains properties that are shared among all objects of that class. The XView class hierarchy is shown in Fig.3.3.

Handles

A *handle* is returned, for a specific object, at the time of its creation. This handle is then used to identify that object, and is passed to the appropriate function to enquire about its state or manipulate it. Handles are opaque in the sense that it is not possible to see through them to the actual data structure that represents the object.

Attribute-Based Functions

The problem of how the client is to manipulate the objects must be carefully analysed in a system such as XView which is based on complex and flexible objects. XView provides a small number of functions, which take a variable-length list of attributes as arguments. For a given call to create or modify an object, only a fraction of all the available attributes will be of interest.

The Notifier Model

The various GUI event models differ in three fundamental ways: how events are distributed, who receives events, and what types of events are supported (Nicholson, 1991). In some GUI's, such as X Window-based, the application registers event handler routines with the system. When events occur, the system calls the appropriate event handler. This means the application has control only when one its event routines is called. The Notifier acts as the controlling entity of user processes. It reads input from the operating system and formats it into higher level events, which it then distributes among the different XView objects. It is the Notifier that calls out the various procedures which the application has previously registered. These are usually designated callback procedures or notify procedures. The advantage of this kind of system is that each component of an application receives only the events the user has directed towards it. XView has a two-layered scheme in which the packages that support the various objects — panels, canvases, scrollbars, etc. — interact directly with the notifier, registering their callback procedures. All the application has to do, in turn, is to register its own callback procedures with the object. The normal sequence of XView applications is to create the various windows and other objects and then register the callback procedures associated (if any) with each of them. Control is then passed to the Notifier, which takes over the task of managing the event-driven environment.

3.5.3 User Interface Management Systems

User Interface Management Systems, normally referred to as UIMS, introduce a separate software layer between the user and the actual code that implements the

GUI (Nielsen, 1993). This term covers multiple meanings and different levels of software services (Coutaz and Balbo, 1991), namely toolkits, frameworks (which will be described in the next section) and interactive software that allows the on-line definition of the interface in a direct-manipulation style. Nowadays, the term UIMS is usually applied to this last type, since the current state-of-the-art of user interface technology makes it possible for the user to define what the interface is to look like (i.e., to select and position the desired widgets) without actually having to write any source code. (Note however that the *functionality* of the interface, i.e., the functions it is to perform, still has to be implemented in a conventional way, i.e., writing code). Examples of this last category are Sun's DevGuide and the Macintosh's HyperCard, where the widgets that compose the interface can be selected from menus and *dragged* to the desired location on the base window.

Other terms employed in the same sense are "user interface development tools" (or "environments"), "user interface toolkits", "user interface builders" and "dialogue management systems" (Nielsen, 1993).

3.6 Frameworks

Although libraries of user interface objects hide much of the complexity of GUI programming, there are still some difficulties stemming from the fact that the user interface remains intertwined with the application (Urlocker, 1989).

X toolkits provide a set of interactive widgets, but no framework for design is provided by X or its toolkits. Large applications tend to become unstructured, non-transparent and tailor-made, and reusability of the interface code is severely impaired. If the interface is meant to interact with complex applications, such as simulation of physical systems, extensive portions of the GUI code may have to be re-written in order to accomodate even minor changes in the physical system or the mathematical models used.

An *application framework* is an interlocking set of objects, according to pre-defined relationships, that provides a shell for application programs (Schmucker, 1988). Frameworks are useful to classify, according to the function they implement, and organize in pre-defined relationships, the several GUI components.

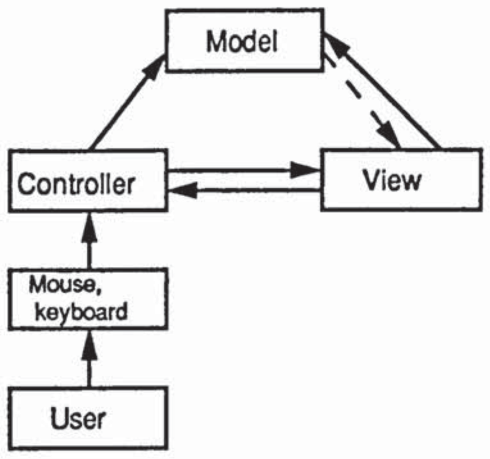


Figure 3.4: The MVC Paradigm.

They reduce the code required in applications, make maintenance easier, and encourage consistency. Different approaches for organizing components in a framework have been described in the literature. Some of the best known ones are the Seeheim model (Green, 1985), the Seattle model (Lantz *et al.*, 1987), the Lisbon model (Duce *et al.*, 1991), the MVC (Model-View-Controller) paradigm (Goldberg and Robson, 1983; Krasner and Pope, 1988; Ayers, 1990) and its more modern successor PAC (Coutaz, 1987a). Application frameworks have also been developed in MacApp (Schmucker, 1986), EZWin (Lieberman, 1985), PPS (Ciccarelli, 1984), GWUIMS (Sibert, 1986), ET++ (Weinand *et al.*, 1988) (based on the architecture of MacApp), InterViews (Linton *et al.*, 1989), Serpent (1989) and Mode (Shan, 1990a, 1990b). Many of these approaches are similar to, or variations of, the MVC concepts. A comparative analysis of several architectures and the PAC model is presented by Coutaz (1987b). One of the latest developments is the Arch model and its generalized counterpart, the Slinky metamodel (Bass *et al.*, 1992).

The Smalltalk language and environment introduced the well known MVC paradigm. In Smalltalk, everything is an object, including models, views and controllers. Applications can be systematically factored into these three components to provide an organizational framework. The components of the MVC triad are related by a *triangular* relationship (see Fig.3.4). Basically, the *model* represents the data and knowledge associated with the problem; the *controller* provides the user with the means to interact with the software system and the *view* enables the graphical display of the output of the model, thus providing interactive feedback (Bourne, 1992).

Relationships between objects can be created in Smalltalk using the *dependency* mechanism; any object can register itself as a dependent of another object. Furthermore, any object may send the message changed to itself, when its state changes. When this occurs, each of its dependents receives automatically an update message. The dependent objects must know how to update themselves, executing their own updating methods. This mechanism is especially important in the relationship between the model and the view, since one or more views are always registered as dependents of a model.

A model can be associated with more than one view-controller pairs. Unlike the model, which may be involved in multiple MVC triads, each view is associated with a unique controller and vice-versa. The view and the controller are tightly coupled; a view's instance variable *controller* points to its controller and a controller's instance variable *view* points to its view. Because both must communicate with their model, each has an instance variable *model* that points to the model object. However, the model does not have a pointer to its view or its controller. This implies that the only way for the model to communicate with the other objects is using the dependency mechanism, i.e., send a changed message to itself. The dependent view(s) and controller(s) respond by querying the model and updating themselves to reflect the change. The view takes responsibility for establishing this intercommunication within a given MVC triad (Burbeck, 1987). Communication among the MVC components is handled by the dependency mechanism and by sending direct messages.

In this work, an MVC-based framework was developed and used. The meaning of each object is somewhat different from the one in Smalltalk and will be described in Chapter 5.

3.7 Evaluation

The best designs start by considering people, not things. (Jacobson, 1992)[1]

Because user interface design is an indefined area, which normally follows an iterative pattern, several authors recommend that it be paired with evaluation, which can used to feed back the design process (Perlman, 1988). Methods used in user interface evaluation can be divided into two major groups. The traditional orientation is empirical, based on measurement and data collection methods, and to a lesser extent, in data analysis. This group of methods is sometimes referred to as laboratory usability testing (Jeffries and Desurvire, 1992). Other methods such as heuristic evaluation (Nielsen, 1993) are intended to augment usability testing, either by being applicable early in the design cycle when usability testing is not possible, or as "discount methods" when resources (money, trained evaluators, etc.) are not available.

The other orientation is based on predictive or theoretical models, which can be viewed as part of the design itself. However, a full understanding of these methods requires an empirical background. Models exist for predicting performance measures such as task completion time, learning time and screen layout.

3.7.1 Usability

Usefulness is the issue of whether the system can be used to achieve some desired goal. It can be divided into two categories: utility and usability (Grudin, 1992). Utility is the question of whether the functionality of the system can do what is needed, and usability is the question of how well users can use that functionality. Usability is thus only one of the factors that determine the system's acceptability (see Fig.3.5). The current draft of ISO standard ISO 9241-11 (guidance on usability specification and measures) defines usability as (Brooke, 1993):

The effectiveness, efficiency and satisfaction with which specified users achieve specified goals in particular environments.

[1]About virtual worlds.

Figure 3.5: Attributes of the acceptability of the system.
(From Nielsen, 1993).

If this definition of usability is used, it is clear that usability is not an absolute value. Rather, it is defined relative to a context, and therefore depends on the situation. Systems and applications are usable or unusable through virtue of being used in appropriate circumstances.

Usability itself is not a single, one-dimensional property of a user interface. It has many components and is normally associated with five usability attributes (Shneiderman, 1992): learnability (ease of learning), efficiency (speed of user task performance), memorability (user retention of commands over time), user error rate, and subjective user satisfaction.

3.7.2 Usability Testing

A usability attribute, as mentioned above, is a general usability characteristic that must be measured as part of the evaluation of a user interface (Hartson and Hix, 1993). A measuring instrument is the method used to provide values for a usability attribute. It can be either objective or subjective, but it always provides quantitative values. Objective measuring instruments provide measures of observable user performance while performing tasks with the interface. Subjective measuring instruments provide measures based on the user's opinion about the interface. Objective measures are commonly associated with a benchmark test, and subjective measures are commonly associated with a user questionnaire. The value to be measured is the metric for which data values are collected during an evaluation session with a participant.

Several such studies have been conducted. Most of these are questionnaire-based, where subjects are asked to rate or in some way give their opinion about several aspects of the interface (Myers and Rosson, 1991; Carr, 1992; Montazemi, 1991; Hix and Schulman, 1991).

3.7.3 Heuristic Evaluation

The "discount usability engineering" method proposed by Nielsen (1993) is based on the use of the following four techniques:

— user and task observation;

— scenarios;

— simplified thinking aloud;

— heuristic evaluation.

Heuristic evaluation is thus the last step in this technique. First, the basic principle of early focus on users should of course be followed. It can be achieved in various ways, including simple visits to customer locations. Secondly, scenarios are a cheap kind of prototype. Prototyping consists of reducing the complexity by eliminating parts of the full system. Horizontal prototypes reduce the depth of functionality and result in a user interface surface layer, while vertical prototypes reduce the number of features and implement the full functionality of those chosen. Scenarios are the ultimate reduction of both the level of functionality and of the number of features; they can only simulate the user interface as long as a test user follows a previously planned path. For any reasonably complex system, the scenario representation is necessarily incomplete (Carroll and Rosson, 1991). Scenarios can be implemented as paper mock-ups or in simple prototyping environments.

The thinking aloud method involves having one test user at a time use the system for a given set of tasks while being asked to think out loud. By verbalizing their thoughts, users allow an observer to determine not just what they are doing with the interface but also why they are doing it.

Finally, heuristic evaluation is done by having an evaluator looking at the interface and emitting an opinion about what is good and bad about it. The goal of heuristic evaluation is to find the usability problems early, in the user interface design, so that they can be attended to as part of an iterative design process. In order to keep the problem manageable, Nielsen (1993) proposes using only ten rules, i.e., ten aspects about which the evaluator must specifically emit an opinion:

1. simple and natural dialogue;
2. speak the users' language;
3. minimization of the users' memory load;
4. consistency;
5. feedback;
6. clearly marked exits;
7. shortcuts;
8. good error messages;
9. error prevention;
10. help and documentation.

3.7.4 Theoretical Methods

The prediction of the usability of the user interface even before testing is appealing to many usability scientists. Such a method would make user testing unnecessary, and also estimate the trade-offs between different solutions without having to build them (Nielsen, 1993). The best known analytic method is GOMS (for Goals, Operators, Methods and Selection Rules)(Card *et al.*, 1983). It involves listing possible user goals (i.e., what the user wishes to perform), the elementary operators available to users (e.g., click the mouse), the methods users compose as sequences of operators, and the selection rules necessary to decide what to do next if the user has several goals pending or if there are several methods that will accomplish the same goal. Obviously, a model of a realistic interface will be huge. Each operation and selection rule is modelled as taking a certain amount of time to carry out, and then the analyst can finally calculate the time needed to perform various tasks by adding up the time for all the individual steps.

One of the few examples of a practical application of theoretical methods is the SANe toolkit (Bōsser, 1994; Bōsser and Melchior, 1992), which is able to produce analytical measures of usability. A SANe model represents the user-visible functionality of a device and the tasks performed by the user, i.e., the interaction model of the device.

In many cases, it is advisable to include more than one technique in the evaluation repertoire (Jeffries and Desurvire, 1992), since all the methods have advantages and disadvantages. Advantages of heuristic evaluation are that it is fast, cheap, and finds a lot of problems. Disadvantages are that it requires multiple evaluations, works best with expert evaluators, and finds a high number of minor problems. Advantages of usability testing are that it overwhelmingly finds severe problems, and it finds problems that affect real users. It has its disadvantages too, the primary ones being cost and that it can only be applied late in the development cycle. The most important limitation of the GOMS model is its limitation to error-free performance by expert users. Modifications to the model are dealing with some of these weaknesses, but due to the need to know a large number of research results and modifications, GOMS and similar approaches are still seen as intimidating by most interface developers (Nielsen, 1993). In equal circumstances, a usability test probably provides the highest quality assessment of an application.

In conclusion, graphical user interfaces are still a major field of research, due to the large number of related issues and the somewhat disparate nature of the two disciplines involved (computer science and psychology). Together with sound software development practices, consideration must also be given to human factors guidelines during the development of a GUI. Formal notations can be useful by making it possible to specify the interface in an implementation-independent manner. Many windowing and graphics systems are available, most of which are non-standard and non-compatible. Evaluation of the interface can be performed during different iterations of the prototype and by several different methods; some are theoretically based whereas others rely heavily on experimental data collection.

In the simulation of complex systems, response time may be so long that it jeopardizes the interactive nature of the application. An introduction to distributed software systems, which enable full advantage to be taken of existing computational resources thus possibly reducing execution time, will be made in the next chapter.

Chapter 4

Distributed Software Systems

4.1 Introduction

A common concurrency problem occurs when several users make simultaneous demands on the computer's processing ability. Current techniques to deal with this situation include multi-tasking operating systems, in which processor time is divided among the several users, and locking strategies in databases (Graham, 1994). A different concurrency situation occurs when two or more related computer processes actually execute simultaneously, which is commonly known as parallel processing. With the development and increasing availability of workstations and efficient networks, distributed software systems, in which related processes can be spread throughout the network and executed concurrently, are becoming common. Although these systems are considerably more complex to program than their centralized counterparts (Achauer, 1993), parallel processing techniques can lead to a dramatic improvement in performance, as reported by Shandle (1990).

In this chapter, an overview of distributed software systems is made.

4.2 Parallel Processing

Increased parallelism may help meet performance requirements of the future (Colbrook, 1993). This is due to the fact that improvements in non-parallel systems have been achieved mainly by the development of better integrated circuits (in that smaller sizes allow for faster processor clock speeds). However, such components are approaching the limits of miniaturisation, arising from quantum mechanics. For very small sizes, electrical behaviour undergoes significant changes and such components can no longer operate on a reproducible basis.

Parallelism can be divided into *data parallelism*, which allows the same operations to be independently performed on different aggregates of data, and *control parallelism*, which allows multiple threads of execution (Agha, 1989). Control parallelism is more general since it implies that each thread of execution may involve distinct data. Control parallel computers can be divided into two broad classes: shared-memory machines and message-passing concurrent computers (multicomputers).

Shared-memory computers have multiple processors, typically 16 to 32, and share a global memory, i.e., the same data space. Shared-memory designs give a price/performance ratio similar to that of PC's (Graham, 1994). The High Performance Fortran (HPF) initiative promises to yield an attractive shared-memory programming model for these architectures (Colbrook, 1993).

Multicomputers use a large number of smaller, programmable computers (processors each with their own memory) which are connected by a message-passing network. The performance of multicomputers with only 64 processors is comparable to that of conventional supercomputers. Software recently developed, which will be described in greater detail in section 4.6, enables the use of separate computers such as Unix workstations, linked by a network, as a single distributed-memory computational resource. In the remainder of this chapter, a distributed-memory paradigm will be assumed.

Parallel processing involves identifying independent tasks to be executed simultaneously. A distinction can be made between "low level" and "high level" parallelism (Vegeais and Stadtherr, 1992). Low level parallelism is associated with fine-grained parallel tasks, at the DO-loop level or lower, that can in principle be identified and executed automatically without user intervention. High level parallelism is associated with opportunities for coarse-grained parallelism that must be recognized by the algorithm developer, based on the knowledge of a specific problem or class of problems, which cannot normally be imparted to a compiler.

Critical to the wide aceptance of parallel processing is the availability of software applications that are used in many branches of science and engineering (Colbrook, 1993). Few are currently available, but as demand for improved performance increases, more are likely to be moved to parallel architectures. Parallel processing techniques may significantly reduce execution times and algorithms are being developed that implement the simultaneous solution of partial problems (Navon and Cai, 1992; Delsanto *et al.*, 1994; Carey *et al.*, 1994) leading to significant reductions in the solution time of the overall problem (e.g., domain decomposition techniques). However, the development of software is still a primary issue, and programming parallel systems efficiently remains a challenge. Not all applications will run effectively on parallel systems, since the problem domain and solution technique chosen must contain a degree of paralellism (Colbrook, 1993).

Objects in OOP, which encapsulate both data and methods, are a very natural unit of distribution for distributed implementations. Furthermore, both object-oriented software systems and distributed software involve the notion of *message passing* (although, in the case of parallel computations, these are messages in the physical sense, as opposed to metaphorical messages in OOP). The analogy is so pointed that object-oriented programming emerges as the natural solution for structuring concurrent software (Graham, 1994).

4.3 Message-passing Mechanisms

Procedure call is a well-understood mechanism for the transfer of both control and data in a single address space. In distributed-memory systems, which rely on message passing, the programmer is required to deal with the additional tasks of message packaging and locating the message target.

Message passing primitives can be classified into two categories: synchronous and asynchronous (Gehani and Roome, 1989). In a synchronous message send, the process that sends a message waits (blocks) until the recipient has accepted the message. In an asynchronous message send, the sender may block until the data has been successfully copied into a buffer, and then continues execution without waiting for the recipient to accept the message. The difference between synchronous and asynchronous message receives is similar. Asynchronous communication is more complex to implement than synchronous communication. An asynchronous model requires buffer allocation (and freeing), and then copying the message into the buffer (Gehani and Roome, 1989). In the synchronous model, no buffer allocation is required and the data can be copied directly from the sender process to the recipient.

Only uni-directional inter-process communication is allowed by both synchronous and asynchronous message passing facilities. The "extended rendezvous" model, which is an extension of the synchronous message-passing model, allows bi-directional communication, in which arguments are used to transmit information from the caller to the recipient and back. This model is used in remote procedure calls (RPC) which will be described in the next section.

4.4 The Client/Server Paradigm

There are several models of distributed programming, of which one of the most widely used is the *client/server* paradigm. In this model, the *server* offers services to the network, which can be acessed by the *client*. In other words, servers provide resources, whereas clients consume them (Corbin, 1991). An application can behave simultaneously as a client and a server. Clients and servers can be either computers or processes.

4.4.1 The RPC Library

The RPC (Remote Procedure Call) library uses the client/server model (Corbin, 1991). In RPC, the familiar procedure call abstraction is provided. Communication between processes is made by making a procedure call, although the procedure may not reside in the same address space of the calling process; the remote service can be accessed and used as if it were a set of procedures in the local program.

The local thread of control is passed over to the remote computer where the procedure is executed and then back to the local application when the procedure returns (see Fig.4.1). Most of the details of control and data transfer are hidden from the programmer (Birrell and Nelson, 1984). However, the pro-

Figure 4.1: Graphical representation of a remote *jump to subroutine* call. (From Corbin, 1991).

grammer is still responsible for locating the target of the call, i.e., the machine or process. Furthermore, the local and remote computers may have different integer or floating point representations. This means that the remote procedure may not interpret properly the values passed to it, the same happening to the local application concerning the values returned. The programmer must therefore, in addition, develop routines in which the conversion of all the arguments to a specific remote procedure is performed, from the local format into a machine-independent format (eXternal Data Representation – XDR) and vice-versa. These routines must be used to code the values sent and to decode the values returned back into the local format. The individual conversion routines for each primitive data type are available in the XDR library, together with the RPC routines.

Synchronous Distribution

The most conservative distribution proposal is built using synchronous communication between sequential processes. An example of this model is the basic *jump to subroutine* call in RPC (Corbin, 1991). A procedure is a routine that takes input arguments, performs processing and returns values to the caller. If a remote procedure is invoked on another machine, the process running on the local machine will not resume processing until the procedure returns. The client sends an RPC request and then waits for the server's reply (or times out waiting for it).

Asynchronous Distribution

When a behaviour closer to one-way messaging, or an asynchronous behaviour is desired, RPC provides means to get around the basic synchronous communication scheme. Three facilities can be used: non-blocking RPC, callback RPC, and asynchronous broadcast RPC. *Non-blocking RPC* can be used when a simple one-way message passing scheme is needed, and is in practice implemented by setting the timeout for the call to zero. If a reply is required, the client must make other arrangements, such as callback RPC, to get the results. *Callback RPC* allows fully asynchronous RPC communication between clients and servers by enabling an application to be both a client and a server. In order to initiate a RPC callback, the server needs a program number to call the client back on. The client registers the callback service, and the program number is then sent as a part of the RPC request to the server. When the server is ready to do the callback RPC, it initiates a normal RPC request to the client, using the given program number. The client must be waiting for the callback either explicitly or via a call to a custom routine that processes incoming RPC requests. Note that *callback* is not being used here in the sense of *event handler* attached to an object, as in Chapter 3, section 3.5.2. Finally, in *broadcast RPC*, the client sends a broadcast packet for a remote procedure to the network and waits for numerous replies. Broadcast RPC treats all unsuccessful responses as garbage by filtering them out.

4.4.2 Client/Server Architecture of X Window

The client/server paradigm is also the heart of the X Window system and forms the basis of the device-independency of X applications. Any X Window system performs two tasks: running the user application and handling the graphics (Reiss and Radin, 1992). *Clients*, in this case, are application programs, whereas *servers* are the display units. Clients communicate with servers over the network (see Fig.4.2)

Figure 4.2: Client/Server architecture of the X Window system. (From Reiss and Radin, 1992).

The client

The client is an application program that makes requests to the server to draw windows, text, and other objects. X Window clients do not communicate directly with the user. Inputs to the application, such as a keypress or a click of a mouse button, are sent to the application by the server. The client then executes X Window commands which in turn request the server to draw graphics. A server may be attached to several clients, which is the reason why output from applications running in several machines can be displayed in the same display unit. The display unit itself can be composed of several physical screens.

The server

The server program runs on each workstation, drawing the required objects on the display. As mentioned before, the server passes user input to the client and decodes client messages (such as the instruction to move a window on the screen). It also maintains complex data structures which reduce client storage needs and diminish the amount of data transmitted over the network.

Each workstation has its own server, which contains the hardware-dependent drivers for that workstation. An X server controls not only the screen but also the keyboard and a pointing device with up to five buttons (Pountain, 1989).

The link

X Window uses currently available networking protocols to transfer data between the client and the server, such as TCP/IP, DECNet, and STREAMS. X Window developers need not know the protocol actually used as X was designed to make use of protocols transparent to the user. Specialized software is needed to send appropriate data and control bytes between the client and the server, i.e. there must be a formal definition of the data stream between clients and the server. The client communicates with the server by sending packets of instructions conforming to the X protocol, which is, in effect, a high-level graphics-description language (Pountain, 1989), similar (as mentioned in Chapter 3, section 3.5) to machine language. The bulk of the X Protocol is asynchronous (Miller, 1990). Data is buffered in both directions and is processed by the receiving end at "some point in the future". To aid debugging, X applications can however force their server connection to be synchronous.

X Protocol Messages

X Protocol messages take one of four formats: request format, reply format, error format or event format.

1. Requests are one-way protocol messages that do not require a system reply. Xlib functions called by the application are translated by Xlib into a protocol request message in the appropriate format, and the message is stored in a special memory buffer. When the buffer is full, Xlib transmits all the accumulated requests. The application continues processing and sending additional requests without waiting for a reply from the server.

2. Replies are used when the application cannot proceed without a specific reply from the server; processing resumes only when the server returns the required information. These special requests are known as *round-trip protocol request messages* and force the client-server communication to become temporarily synchronous; as such, they can cause considerable delay.

3. If a given request causes an error, an X protocol error report is generated and printed. When an X application has enabled synchronization of its connection

to the server, requests are not buffered. The system accepts a request, blocks processing, transmits the request to the server, and waits for a reply. Although synchronization makes it easier to determine the function that caused the error, it slows down performance drastically (Reiss and Radin, 1992).

4. The X server recognizes many (33) event types including pointer motion, key press, button press and release, window entry and exit, input focus switching, exposure of previously covered windows, etc.. Events are usually packets sent to clients by the server, asynchronously, to notify them that something has happened in or to a window (Miller, 1990).

Available on practically all Unix-based platforms, X manages input and output to the display of the workstation both locally and across a heterogeneous networking environment. The server encapsulates all device dependencies; the communication protocol between clients and servers is device independent. If a different display type is used, only a new server implementation must be performed, and no changes in the application itself are required (Pountain, 1989). Applications are thus device-independent as mentioned in Chapter 3, section 3.5. At present X Window is the only windowing system that really works in a multiuser, multicomputer, networked environment, allowing, for example, windowing software to be run on a supercomputer and the result to be seen on a personal desktop machine.

4.5 Object-oriented Concurrent Distribution

The importance of object-oriented concurrent programming, still a new field for researchers, is assured by the fact that distributed and client/server architectures must involve concurrency for an effective use of resources. Furthermore, no other programming style offers a clear model for such a complex, cooperating concurrent system (Graham, 1994).

The optimal extension to object-oriented software that can address the needs of concurrent and distributed computing as well as those of sequential computing is still unclear. Object-oriented concurrent systems, which support object-orientation, concurrency and dynamic adaptation, are very complex (Yoon, 1992). A large number of languages and compilers has been developed in recent years, although the survival of some of them is yet to be determined. In this section, a brief overview of related languages is made.

4.5.1 Message-passing Languages

Message-passing languages can be divided into object-oriented languages and actor languages (Carré and Cléré, 1989). Although both rely on the same general ideas, they do not offer the same functionality to the programmer. The common idea to both these languages is that an application is designed as a set of entities that communicate by messages, hence the general denomination *message-passing* languages. Each entity has a prescribed behaviour that describes how it reacts when it receives a message.

Other analogies exist between objects and processes, or more accurately between the underlying abstractions: classes and process types (Meyer, 1993). Both categories support local variables (attributes of a class, variables of a process or process type), persistent data (which keeps their value between successive activations), encapsulated behaviour (a single behaviour for a process, any number of methods for a class), and restrictions on how modules can exchange information.

From these basic rules, research has diverged in different directions (Carré and Cléré, 1989). An important difference between actors and object-oriented languages is how behaviour is defined. Whereas the behaviour of an actor is defined in itself, i.e., actor systems do not possess the property of inheritance, the behaviour of an object is normally obtained from its class and superclasses. A system of actors can be seen as a set of relatively independent entities; the only relationship is a dynamic one created when a communication occurs between two actors via asynchronous message passing (Agha, 1990). Conversely, an application designed with an object-oriented language builds a set of entities linked together by a dynamic and complex graph, due to all the connections between objects: instantiation, inheritance, possibly dependency dictionary, etc.. Actor languages offer a simpler model of communication, whereas object-oriented languages offer better structural properties. The distribution of objects among the different elements of the architecture offers many problems in object-oriented languages, due to the many links between the objects. No solution has been found so far that allows efficient execution of object-oriented distributed applications. It can be doubted whether this approach is able to live in a distributed environment (Carré and Cléré, 1989).

Actors Languages

In the actor paradigm, the universe contains computational agents called actors, which are distributed in time and space (Agha, 1989). The actor model of computation is based on the idea that concurrent systems in the world can be modelled as systems of objects known as actors (pools of cooperating workers or experts), acting only on local information and interacting solely through message passing (Manning, 1989). An actor has defined responsibilities, needs and knowledge about collaborators (Graham, 1994).

Actors have unique permanent identity. Each actor has a local state, a location (its mail address) and a behaviour. The behaviour defines what it will do when it receives a message and can be divided into a *script*, i.e., the instructions which the actor must follow (similar to a *method*), and the acquaintances, i.e., the mail addresses of other actors the actor knows about (similar to *instance variables*). Acquaintances determine the other actors with which the actor may communicate; an actor with no acquaintances is a candidate for garbage collection.

When an actor receives a message, it reacts to it using the information available. It may make simple decisions, create new actors, send messages to the other actors it knows about, or specify a replacement behaviour (Agha, 1990). If it does not have a method to handle the message, the actor may *delegate* to a *proxy*, which is an actor nominated among its acquaintances. No methods are inherited in delegation, i.e., no code is copied; all that takes place is message passing. Some or all of these actions may occur concurrently.

The only way one actor can influence the actions of another actor is to send the latter a communication. The integrity of the two communication units must be maintained, implying that there must be some mechanism for serializing incoming communications. This can be achieved by requiring a sender to wait until the recipient is free to accept a communication and then blocking the recipient from accepting any other communications until it has finished processing the first communication.

Most actor-based and concurrent object-oriented languages provide more than a collection of objects which communicate via message passing. Distributed techniques for resource management, object addressing, garbage collection, and computation management must be addressed, especially if objects are dynamically created and discarded as the program executes (Bensley *et al.*, 1989).

Some actor-based languages have been developed as extensions to Smalltalk, by creation of new classes. Examples are Actra (Thomas *et al.*, 1989), where an object from the class *actor* encapsulates a community of objects intended to execute concurrently with other communities. Another actor-based extension of Smalltalk is Actalk (Briot, 1989), where an actor is built from a standard Smalltalk-80 object by associating a process with it and by serializing the messages it can receive into a queue.

PLASMA II (Lapalme and Sallé, 1989) (an extension of Plasma, the first actor language originally proposed by Carl Hewitt), Acore (Manning, 1989) and Cantor (Athas and Boden, 1989) are other examples of actor-based languages. In general, actor languages are very low-level and difficult to use (Graham, 1994).

Message Driven Computing (MDC) is a generalization of actors. It is based on two fundamental characteristics, namely the fact that messages, not sequential processes, convey both control and data (there are no sequential processes unless they are programmed explicitly in terms of messages) and computation is invoked by the presence of a collection of messages at the same location. It is possible to implement an actor system using MDC; for further details, see Christopher (1989).

Other Languages

Integrating distribution and object-oriented languages has attracted much attention in recent years (Achauer, 1993). However, the evidence that concurrency blends well with the object paradigm comes mainly from systems which are object-based, not object-oriented; as mentioned before, inheritance is rarely involved (Lõhr, 1993). Different degrees of integration may be considered when comparing object-oriented languages that support concurrency:

1. no integration: the object-oriented features and the concurrency features are independent of each other. Although such a language is both object-oriented and concurrent, it cannot be considered a true concurrent object-oriented language (COOL). An example is concurrent C++, which results from the fusion of two separately developed extensions of C: Concurrent C (Gehani and Roome, 1989) and C++ (Stroustrup, 1986).

2. partial integration: concurrency is integrated into the object-based language, allowing the existence of synchronized objects, active objects, and asynchronous object invocation. Inheritance, however, does not apply to active and/or asynchronous classes. Such a language is not a pure COOL either; it could be termed "concurrent object-based with inheritance". An example is GUIDE (Decouchant *et al.*, 1989);

3. full integration: a real COOL supports inheritance hierarchies that include active classes. An example is POOL (Parallel Object Oriented Logic) (Koegel, 1989).

Furthermore, several degrees of concurrency are possible. Objects can be classified as single thread objects, objects that may service only one request at a time but may create multiple threads internally for servicing it, and objects that may service multiple requests concurrently (Papathomas, 1989). Note that increasing synchronization support must be provided with an increasing degree in concurrency.

Numerous proposals have tried to combine concurrent programming with object-oriented programming. Three distinct approaches for introducing concurrency to object-oriented systems can be identified (Karaorman and Bruno, 1993), namely designing a new concurrent object-oriented language, extending an existing object-oriented language, or, finally, using an existing object-oriented language, provide concurrency abstractions through external libraries. Most of the earlier systems take the first approach, i.e., designing a new object-oriented language with built-in concurrency. The extensions to existing sequential object-oriented languages introduce concurrency using some combination of the following techniques:

1. inheritance from special *concurrency classes* that the modified compiler recognizes. An example is Parallel Eiffel (Eifell//) (Caromel 1989, 1993);

2. special keywords, modifiers or preprocessing techniques to modify or extend the language syntax and semantics. An example is CEiffel (Lõhr, 1993).

Finally, concurrency can be introduced by the creation of new libraries, namely a class definition of Process. Since the latest trends for object-based concurrency emphasize issues such as reusability and compatibility, the library-based solutions are attractive, as they do not replace the existing software development platform. The library approach is the most recent one and has been influenced by most of the earlier work on concurrency (Karaorman and Bruno, 1993).

Distributed concurrent object-oriented programming involves more than objects statically residing in several address spaces; when objects are allowed to reside on multiple nodes, ways to address and locate them must be provided (Jazayeri, 1989). Their migration from one node to another at run-time is also a desirable feature. Furthermore, the amount of message traffic is crucial to the efficiency of the system and must be minimized. Also, if a set of objects is responsible for managing a distributed data structure, it is important to be able to change the distribution of the data among the member objects, since the right distribution leads to higher performance. Finally, once multiple threads of control are allowed, synchronization and concurrency control must be supported.

Some systems that offer both location-independent invocation and object migration are Emerald (Black *et al.*, 1987), which is a strongly typed, object-based language, Amber (Chase *et al.*, 1989), which augments a subset of C++ with primitives to manage concurrency and distribution, and DOWL (Achauer, 1993), which is a proxy-based (see section 4.5.2) extension of the Trellis object-oriented language.

Many other concurrent object-oriented languages have been developed in recent years, such as DOCASE (Mühlhäuser *et al.*, 1993), Interwork II (Bain, 1989), Heraklit (Hindel, 1989), MACE (Gasser, 1989), and MELD (Kaiser, 1989). Karaorman and Bruno (1993) propose the introduction of concurrency to the object-oriented language Eiffel through a set of class libraries. No changes

Figure 4.3: An invocation on a remote object is intercepted by the *proxy* object. (From Steele, 1992a).

are made to Eiffel or its run-time system. Although sometimes this is not explicitly stated, some of these languages rely on concepts which are very similar to actor concepts.

4.5.2 The *Proxy* Paradigm

The object-oriented programming paradigm, with its notion of synchronous message passing between distinct objects, maps neatly onto the client/server model of distributed systems (Steel, 1992a). The object-oriented and client/server paradigms can be combined to provide a distributed object model. A mechanism similar to RPC can be used, *remote method invocation* (RI), to provide transparency to the programmer.

DPS is a prototype system based on an implementation of Smalltalk, BHH Smalltalk (Xu and Dollimore, 1992). If the implementation of a remote operation invocation has the same interface as the local operation invocation, it is a transparent remote operation invocation. The transparency is achieved by automatically providing a local *proxy* for each remote object upon whom operations can be invoked by a local object. The function of a proxy is to behave like a local object towards the message sender, but instead of executing the message, it forwards it to the remote object, thus hiding the complexity of inter-machine communication from the user (see Fig.4.3). Proxies behave exactly like the object they represent (Achauer, 1993). The proxy blocks the local activity thread until the result of the remote invocation is returned. The operation is then performed at the object's actual location. Any values or exceptions raised are transmitted back to the proxy, which then returns the result, just as in local invocation, so that the local activity thread continues. The RI model can be viewed as keeping the data (or objects) in a fixed place and moving the thread of control to where the data resides. This mechanism therefore provides a *sequential* type of object-oriented distribution.

Although Remote Invocations encapsulate the complexity of inter-process communication, RI suffer from being inherently synchronous, since the calling task cannot continue until it has received a reply from the called procedure. When results are not required, or a request needs to be multi-cast to several different receiving tasks, RI is not an ideal model.

4.6 Inter-process Communication Enabling Software

PVM (Parallel Virtual Machine) is a recent software system which enables inter-process communication. A user-defined collection of serial, parallel and vector computers can be made to behave as one large distributed-memory computer (Geist and Sunderam, 1993; Geist *et al.*, 1994), referred to as the *virtual machine.*

PVM supplies functions to automatically start up tasks (units of computation analogous to Unix processes) on the virtual machine and allows the tasks to communicate and synchronize with each other. Applications, written in Fortran77 or C, can be parallelized using message-passing constructs common to most distributed-memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel. PVM supports heterogeneity at the application, machine and network levels. It handles all data conversion that may be required if two computers use different integer or floating point representations, thus avoiding the need for lengthy programmer-defined routines to convert arguments to and from their XDR representations.

The PVM software is composed of two parts. The first part is a daemon that resides on all the computers making up the virtual machine. The second part is a library of user callable routines for spawning and coordinating tasks, passing messages between them (any task can send a message to any other PVM task), and modifying the virtual machine (add and delete hosts). There are routines to send signals to other PVM tasks, and find out information about the virtual machine configuration and active PVM tasks. It also includes routines that enable a user process to become a PVM task and to become a normal process again.

Synchronization is ensured since PVM provides asynchronous blocking send, asynchronous blocking receive and non-blocking receive functions. In PVM's terminology, a blocking send returns as soon as the send buffer is free for re-use regardless of the state of the recipient. A non-blocking receive returns immediately with either the data or a flag that the data has not arrived, and a blocking receive function returns only when the data is in the receive buffer. The PVM model guarantees that message order is preserved. The maximum size of the messages that can be sent or received is limited only by the amount of available memory on a given host.

Note that PVM is not a distributed object-oriented system, nor does it enforce any distribution paradigm. However, the transparency with which it provides inter-process communication and task management is an important feature and allows the programmer to implement an adequate concurrent paradigm with greater flexibility than other systems. Chapter 5 describes the use made of PVM in this work.

4.7 Distributed Operating Systems

As mentioned in previous sections, many attempts have been made to extend existing object-oriented languages with support for distributed objects by adding message-passing facilities. However, this extension is in general an inefficient approach (Lea *et al.*, 1993) due to the fact that traditional operating systems were designed before networking became commonplace and hence provide little support for inter-process communication (Steel, 1992b). Most of them provide abstractions that were not designed to support modern programming languages or distributed applications, treating each process as an independent entity for the purposes of resource management (Lea *et al.*, 1993). For distributed applications, spanning multiple address spaces, the compiler support breaks down because it is not aware of the environment outside a single address space. Also, some languages support lightweight activities or active objects, whereas most systems support a heavier notion, a process. Finally, current operating systems provide distributed interprocess communication using protocols designed for unreliable networks as an "add-on" feature. These communication mechanisms are often too costly to support applications built of fine-grained objects that use a large number of inter-object invocations.

Some recently developed architectures and operating systems aim to directly address these problems. The Rosette architecture (Tomlinson *et al.*, 1989) is an actor-based architecture formed by two major components, namely an interface layer and a system environment. The interface layer includes a set of actors that represent the processing, storage and communication resources. The system environment contains actor communities which provide monitoring, debugging, resource management, and compilation facilities.

Choices is an object-oriented operating system built in C++ (Campbell *et al.*, 1993). The advantages of C++ are that it is efficient, portable, and available on a variety of platforms. Objects are used to model both the hardware interface, the application interface, and all operating system concepts including system resources, mechanisms and policies. High level features that are available in other less efficient languages were built using C++ language primitives, classes, and subclasses. User, server and system objects can be defined, created and deleted dynamically.

Another solution to this problem is the use of microkernel architectures as CHORUS, Amoeba and Mach, which provide a basic set of abstractions designed to allow programmers to build operating systems (Lea *et al.*, 1993).

The practical acceptance of any of these approaches is yet to be determined.

To summarize, parallel processing is at present an answer to the ever increasing demand on computer power. The feasibility of combining concurrency with object-orientation is still under study. Most software systems of this type are object-based (e.g., actor languages) rather than object-oriented, due to the difficulty of transferring the many links between objects and classes into a distributed environment. However, actor languages are low-level and difficult to use. Software like PVM allows the introduction of concurrency into higher-level languages such as C++, which in turn have immediate access to high-level user interface software such as the X Window-based toolkits. This approach was therefore selected in this work, where the construction of sophisticated user interfaces is sought, and will be described in the next chapter.

Chapter 5

Interactive Object-oriented Simulation

5.1 Introduction

The activity of building models of the real world and simulating them on a computer is usually referred to as *modelling and simulation* (Zeigler, 1976). Three main elements are involved in this process: the real system, the model, and the computer. To model a system is to replace it by something which is simpler and/or easier to study, and yet equivalent to the original in all important aspects (Mitrani, 1982). Modelling and simulation may serve several purposes, namely to understand how the real system behaves, to optimize certain aspects of its operation, or because experimentation with the real system is costly, time consuming, or impossible. Furthermore, computer experiments are completely repeatable and non-destructive; they can be re-started at any point without destroying the subject of the study.

However, simulation practices in the scientific community have not accompanied the evolution in hardware performance and in software development concepts (Peskin *et al.*, 1989). Computers are primarily used as numerical production tools, instead of tools for interactive prototyping. In spite of the recent (and increasing) importance given to graphical user interfaces, most simulation programs still do not benefit from an interactive environment or the application of object-oriented concepts. Consequently, decision makers are unable to explore and solve complex problems in an interactive and graphical environment (Armstrong and Densham, 1992).

Object-oriented concepts are invaluable for the development of structured interactive software. In the special case of interactive simulation, OO concepts provide support at three levels: firstly, they are essential for the definition and manipulation of the widgets that compose the interface. Secondly, they make it possible to retain a clear separation between the domain-specific component and the graphical user interface, by utilization of an adequate framework. Thirdly, in what concerns the simulation itself, the elements that compose the system being simulated can be treated in a fully modular fashion, i.e., as objects, which is essential for the creation and deletion of objects at run-time, and for adaptability to changes in the configuration of the system.

In this chapter, the main issues related to the implementation of interactive simulation of complex systems are identified and discussed. Solutions are proposed and the basic principles and features of the working prototype developed in this work are presented, namely the framework used for design and the type of distributed approach implemented. The prototype is further described in greater detail in Chapters 6 and 7.

5.2 Mathematical Modelling

Models can be classified into several categories, according to the basic assumptions used (Zeigler, 1976). Common to the simulation of all types of models are the notions of *events* and the need to keep an internal *clock*. The system state changes in the course of an operation path; these changes of state are called "events", which occur at certain times ("event times"). The passage of time is represented by incrementing the value of the internal clock.

The most immediate classification relates to the *time base* on which model events occur. A model is a *continuous time* model if time is specified to flow continuously, i.e., the clock of the model advances smoothly toward ever-increasing values. A model is a *discrete time* model if, in contrast, time flows in jumps, i.e., the clock advances periodically, jumping from one value to the next.

A second classification is related to the type of values assumed by the variables that describe the model. The model is a *discrete state* model if its variables assume a discrete set of values; it is *continuous state* if their ranges can be represented by real numbers (or intervals of these) and *mixed state* if both kinds of variables are present.

Continuous time models can further be divided into *discrete event* and *differential equation* classes. A model specified by differential equations is a continuous time - continuous state model in which state changes are continuous, i.e., both the effects of the events and the intervals between event times are infinitesimal. The time derivatives (rates of change) of the variables are governed by the differential equations. If more than one independent variables exist in a differential equation, it is classified as a *partial differential equation.* Systems described by this type of models are called *distributed-parameter systems* (Karplus, 1977). If the model is formed by a set of difference or ordinary differential equations, these systems are referred to as *lumped-parameter systems* (Karplus, 1977). In both these cases, the simulation consists of constructing one or more functions that uniquely satisfy the differential equations for given initial and boundary conditions; *integration* is the heart of continuous-system simulation. This type of continuous modelling has been used extensively in this study and is one of the primary concerns of this work.

In a continuous time - discrete event model, even though time flows continuously, state changes can occur only in discontinuous jumps. A jump is triggered by an event, and (since time is continuous) these events can occur arbitrarily separated from each other. An operation path is completely determined by the sequence of event times and by the discrete changes in the system state which take place at these moments. In between two consecutive event times, the system state may vary continuously.

In discrete time models, the system is considered only at selected moments in time (the operation paths of these models are sequences of system states, typically evenly spaced in time); any changes of state are noticed only at observation points. It is therefore assumed that events in the real system are allowed to occur only at certain moments. Continuous systems can be approximated by a discrete time model to any degree of accuracy by choosing a sufficiently small time increment. In discrete event systems, the simulator must keep track of the passage of time, generate the events that change the system and implement the resulting changes.

A third classification is related to the incorporation of random variables in the description of the model. In a *deterministic* model no such random variables appear. Differential equation models are essentially deterministic in nature. A *probabilistic* or *stochastic* model contains at least one random variable. The incorporation of random variables in the model may, or may not, reflect the absence or presence of random phenomena in the system being modelled. Both discrete time and continuous time - discrete event models are normally stochastic models.

After the model has been defined, the solution can be obtained in a number of different ways, namely using analytical or numerical methods (Mitrani, 1982). Analytical solutions provide a closed-form expression for the desired system characteristics. Such solutions, while clearly advantageous, can usually be obtained only for the simplest models. Numerical solutions can, in principle, be applied to models of arbitrary complexity, especially as the processing power of digital computers increases. They have however the disadvantage of producing results only for isolated points in the domain.

The behaviour of the clock of a simulator may be very similar to a real one: at every tick, the value of the clock is incremented by a given (constant) amount. Such simulators are called "fixed time increment" or "synchronous". As mentioned above, it is always possible to simulate the passage of real time by making very small, fixed increments in the model's time variable and changing the system as required at each increment. This tends to a "quasi-continuous" view of the world and is the basic approach to continuous systems simulation. However, this is not necessary for a discrete event system; the clock may be incremented directly from one event time to the next event time, taking on each occasion the appropriate actions, regardless of the time interval that separates the events. Simulators of the second type are called "variable time increment" or "asynchronous". The choice of simulator structure depends on the nature of the system being simulated.

5.3 Simulation Languages

Analogue computers were originally used for the simulation of continuous models (Kreutzer, 1986). In more recent times, the operation of these hard-wired machines has been emulated on digital, stored-program computers, with a considerable gain in programming convenience and computational accuracy. *Continuous simulation languages* evolved from attempts to replicate the functioning of analogue computers. In this section, a brief overview of the languages for continuous simulation will be made.

The IFIP Conference on Simulation Programming Languages in 1968 was a major milestone in the history of simulation; since then, the development of new systems has progressed at a much slower rate. The first special-purpose languages for continuous-system simulation were Fortran-based; in fact, this is still the case nowadays (Kreutzer, 1986). Because of the number-crunching nature of numerical solutions to differential equations, run-time efficiency and accuracy are overriding concerns. Most research is consequently targeted on these areas. Fig.5.1 shows the lineage of some of the best known systems. The best known discrete-event simulation languages are probably SIMSCRIPT II.5, GPSS and SIMULA (Mitrani, 1982). For an overview of the evolution of the most important families of *discrete-event* simulation languages, see Kreutzer (1986).

Primitives for dealing with concurrency, creation of dynamic objects, and handling of dynamic relationships between them are not well supported by most of the widely-used programming systems. Fortran and Fortran-based simulation packages have several deficiencies, namely the fact that arrays and routines are the only structuring devices, only static storage management is allowed (which requires compile-time dimensioning of all data structures) and names up to six characters only can be used. Although at first these limitations may appear to be minor inconveniences, they in fact prove to be severe handicaps for writing readable and well-structured programs (Kreutzer, 1986). These deficiencies grow worse at an exponential rate as model complexity increases. Eventually, the source code consists of a tangled mesh of variables representing entities and links between them, event routines and scheduling references (Kreutzer, 1986). Such

Figure 5.1: Origin and evolution of some continuous-system simulation languages.
Combined discrete/continuous languages are underlined. (From Kreutzer, 1986).

models are very difficult to understand, communicate, debug and verify. They are obviously totally inadequate for graphical interactive environments, where the definition of the physical system is done at run-time, usually by choosing the desired components from menus. The only advantage that can be attributed to Fortran-based simulation systems is their wide availability and portability. Fortran has served its purpose well, but it was designed for an environment characterized by batch operation, expensive processor and memory resources, and an absence of any guidelines for the design of good cognitive tools; the computing scene is now radically different. Kreutzer (in 1986) considers Fortran to be an outdated tool.

Since a major requirement for a simulation language is entity manipulation (creation of new entities, destruction of old ones, placement of entities into and removing them from ordered and unordered sets, etc.), the object-oriented programming style is particularly appropriate for the design and implementation of simulation programs. As the pioneer of object-oriented programming languages, SIMULA (Dahl and Nygaard, 1966) deserves here a special reference. The first version of SIMULA, SIMULA I, was especially designed as a tool for discrete-event simulations. SIMULA 67, the current version, has been totally redesigned and has become a powerful general-purpose language, which contains a slightly modified ALGOL60 as a subset. A major addition to ALGOL was the class concept. Two pre-defined classes (classes SIMSET and SIMULATION), especially dedicated to simulation purposes, were built into the system. Class SIMSET supplies entity and set manipulation facilities, including linked lists, and class SIMULATION supplies a clock routine and an event list, manipulation of parallel processes, and timing and scheduling primitives. Various random number generation procedures are provided. Also, SIMULA supports co-routines as one of its main control structures. Co-routines implement a less restrictive control paradigm than the master-slave paradigm, used in conventional function and procedure calls.

5.4 Development of Interactive Simulation Software

In spite of the benefits provided by the application of object-oriented concepts, OOP has not seen widespread use in the domain of the simulation of complex systems, mostly due to the absence of powerful object-oriented packages of numerical methods. This is especially true in the case of continuous time-continuous state models, which are of interest to this study. The mathematical models that describe this type of system may include sets of ordinary differential equations (ODE's), partial differential equations (PDE's), differential-algebraic equations (DAE's), linear or non-linear algebraic equations, evaluation of integrals, etc.. Most mathematical models for systems of interest in engineering areas, for example, fall in this category, such as the lime kiln of a paper pulp plant (Pais and Portugal, 1993a, b) or the simulation of stresses and strains in a structure, such as a bridge withstanding traffic (Kimbrell, 1989). Traditional procedural simulation programs have the form of calls of specialized and usually lengthy procedures available in standard libraries such as NAG or specific solution routines such as LSODI, DDASSL, DRKMXX, CONLES, JCOBI, DFOPR, etc.. Most or all of these procedures are written in high-level procedural languages such as Fortran77 or C. Considerable research efforts are now being devoted to the OOP implementation of advanced numerical methods, such as the linear and non-linear finite-element methods (Zimmermann *et al.*, 1992; Dubois-Pèlerin *et al.*, 1992; Menétrey and Zimmermann, 1993), where an analysis of the algorithm is made in order to determine the adequate class hierachy. Full general-use libraries of object-oriented numerical methods do not exist yet.

Another problem associated with interactive simulation lies in the fact that very complex problems may not be suitable for on-line simulation due to the execution time, which can lead to unacceptably long response times. Even if object-oriented numerical methods exist for the problem in question, this is not a guarantee of faster execution. Indeed, and although Smalltalk has been proposed to implement simulation studies of simplified industrial units (Lazarev, 1991), execution may take considerably longer if essentially interpreted languages like Smalltalk are used. Parallel processing techniques can solve complex models at high speed, therefore significantly reducing response times, and enabling decison makers to quickly see the results of revising parameters and criteria (Armstrong and Densham, 1992). As parallel programming environments become commonplace in the coming decade, the need for suitable parallel versions of sequential algorithms will increase. Software (e.g. the PVM libraries) has also been recently developed that enables inter-process communication and the use of networked machines, such as UNIX workstations, as a single distributed-memory, multiprocessor machine, thus making it possible to use commonly available network configurations as a multicomputer.

Finally, several advantages are provided by the separation between the graphical user interface and the domain-specific component (Dodani *et al.*, 1989). The interface can be altered without repercussions on the application code. It can also be further subdivided into smaller, re-usable components, which may be used without a detailed understanding of the underlying implementation. This facilitates the modification of the interface in order to be used in other applications. This approach also allows the interface to be developed in an iterative manner, where successive prototypes are produced until a satisfactory one is completed; successive iterations can be coupled to the same domain-specific component. This separation can be achieved by application of an adequate framework, where the overall application is systematically factored into predefined components linked by pre-defined relationships.

In this work, a prototype was developed which aims to address the main problems associated with interactive object-oriented simulation. C++ was used as the programming language for the graphical user interface. This choice was made due to several reasons, namely its object-oriented features, efficiency of execution of the compiled code, simple interface with Fortran77 (in which the numerically-intensive modules were written) and, since the windowing system selected was X Window, due to the advantages pointed out in chapters 3 and 4, immediate access to X Window and X-based GUI toolkits (specifically, the XView toolkit). Sun's SPARCompiler C++ 3.0.1 and SPARCompiler Fortran 2.0.1 were used, running under the Solaris 2.3 Unix operating system. The windowing environment was Sun's Openwindows V.3.3, implemented on top of X Window V.11 (X11) using the XView toolkit. A Model-View-Controller-based framework was developed for the interactive application. Concurrency was introduced to the C++ language, in a limited form, by means of external libraries (PVM libraries, see section 4.6).

5.5 Decomposition of the Overall System Used in the Prototype

The need for the decomposition of the overall model into several submodels for the solution of complex systems has been recognized and described for several types of problems, among which chemical process flowsheeting (Vegeais and Stadtherr, 1992). The modular approaches consist of dividing the overall model into modules, or sets of smaller equations, which can then be solved either sequentially or simultaneously. In steady-state calculations such as process flowsheeting, the solution of the individual modules can be repeated as many times as necessary until convergence is achieved. Note that, in this case, decomposition of the system into smaller subsystems is used as a numerical technique and does not lead necessarily to an approximate solution.

If the physical process is continuous in time, as in most dynamic simulations, a straightforward decomposition corresponds to an approximate solution. The transient state of a system is typically described by — or can be transformed into, using finite differences, orthogonal collocation, or other methods — sets of simultaneous ordinary differential equations. In a procedural approach, a vector of unknowns is fed to the solution routine, where the unknowns are computed simultaneously for the specified time step and the results are returned to the main program. Although this approach is accurate — excluding the approximations involved in the numerical methods applied themselves — the flexibility of this type of solution is virtually nil. Extensive code re-writing and debugging has to be done if, for example, more variables must be inserted between existing ones, and this approach is obviously not suited to applications where the system is to be defined on-line.

If an object-oriented approach is taken, then a decomposition of the system into suitable objects is made. Each object must be able to evaluate its own state,

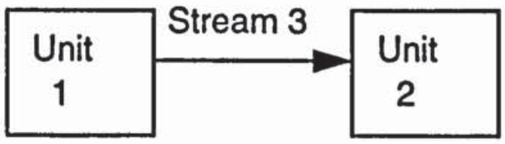


Figure 5.2: Decomposition of the overall model into objects.

according to its instance variables and methods, for the desired conditions. The coarser the decomposition — which corresponds to fewer objects each with a high number of instance variables and complex methods — the closer the final algorithm will be to that obtained by a non object-oriented approach.

5.5.1 The *Unit* and *Stream* Paradigm

The decomposition of the system into *units* and *streams*, which have fundamentally different natures, is proposed in this work. Each *unit* is a data processing object. It possesses instance variables which can be divided into *independent variables* $\overline{x} = (x_1, x_2, ..., x_m)$, *dependent variables* $\overline{y} = (y_1, y_2, ..., y_n)$, and *parameters* $\overline{a} = (a_1, a_2, ..., a_p)$. More elaborate models can include other classifications such as *observed variables*, *manipulated variables*, etc.. The *dependent variables* define the *state* of the system and are evaluated by the object by application of adequate methods. These methods must therefore include all the information needed to solve the problem. Fig.5.2 represents a simple system, where units 1 and 2 are connected by a stream (object 3). If, for example, the mathematical model that corresponds to *unit* 1 is or can be transformed into a set of first order PDE's, depending on one spatial dimension and time ($\overline{x} = (x_1, x_2)$, where $x_1 = z$ and $x_2 = t$) then its updating method must solve the mathematical model defined by

$$f_i(t, z, \overline{y}^1, \frac{\partial \overline{y}^1}{\partial t}, \frac{\partial \overline{y}^1}{\partial z}, \overline{a}^1) = 0, \ i = 1, n_1 \tag{5.1}$$

Initial and boundary conditions must be included and were omitted here for the sake of simplicity. Now, let the updating method for *unit* 2 be such that it implements the solution of the following system of algebraic equations:

$$f_i(\overline{y}^2, \overline{a}^2) = 0, i = 1, n_2 \tag{5.2}$$

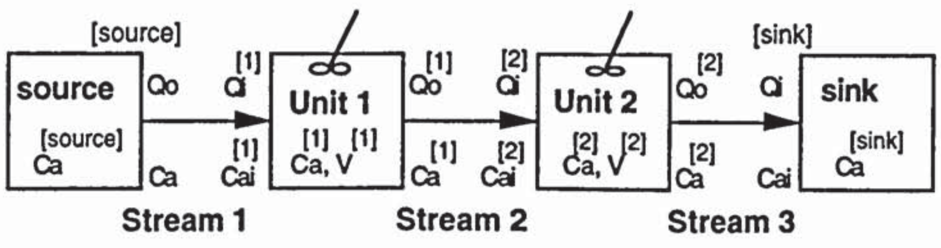


Figure 5.3: Example of decomposition of the overall model into objects (set of two continuous stirred tank reactors).
Ca = Concentration of Species A (mol/m^3)
K = Kinetic Parameter (s^{-1})
Q = Volumetric Flowrate (m^3/s)
V = Volume (m^3)

These objects may correspond each to a physical object, part of it or even a set of physical objects, as desired — the granularity of the decomposition is not fixed. Objects can be related by simple relationships, or by more complex relationships which require themselves some form of calculation. In either case, these are performed by the *stream* objects. Although a *stream* may have some form of connotation with a physical object, it is used here simply as an object that represents a relationship. The updating method for object 3 must include:

$$f_i(\overline{y}^1, \overline{a}^1, \overline{y}^2, \overline{a}^2, \overline{a}^3) = 0, \ i = 1, n_3 \tag{5.3}$$

Each stream object must always have an origin and a destination units. This allows the differentation of the streams, according to the units they link; if necessary, units can be artificially added to the system. For example, if a certain unit is subject to inputs of two different kinds, these can be easily associated with two streams, which have source units of different classes, and the same destination unit.

As an example, let us consider the simplified case of the simulation, in transient state, of a set of two isothermal continuous stirred tank reactors without level control, where first-order reaction occurs (see Fig.5.3).

The updating method for Unit 1 corresponds to the solution of eqs.(5.4) and (5.5) below (initial conditions omitted for simplicity):

$$\frac{dV^{[1]}Ca^{[1]}}{dt} = Qi^{[1]}Cai^{[1]} - KCa^{[1]}V^{[1]} - Qo^{[1]}Ca^{[1]} \tag{5.4}$$

$$\frac{dV^{[1]}}{dt} = Qi^{[1]} - Qo^{[1]} \tag{5.5}$$

Similarly, the updating method for Unit 2 corresponds to the solution of:

$$\frac{dV^{[2]}Ca^{[2]}}{dt} = Qi^{[2]}Cai^{[2]} - KCa^{[2]}V^{[2]} - Qo^{[2]}Ca^{[2]} \tag{5.6}$$

$$\frac{dV^{[2]}}{dt} = Qi^{[2]} - Qo^{[2]} \tag{5.7}$$

The updating method for the source unit can be empty, which means the values of $Ca^{[source]}$ and $Qo^{[source]}$ remain constant in time, or it can possibly re-define $Ca^{[source]}$ and $Qo^{[source]}$. Let us take the updating method for the sink unit to be simply $Ca^{[sink]} = Cai^{[sink]}$. After all the units have updated their state, the update message is sent to the streams to re-establish the relationships between the units. The physical system is thus considered to be composed of 7 objects (4 units and 3 streams). Three unit classes are involved: source unit class, sink unit class, and the class that unit 1 and unit 2 belong to. Three different stream classes are involved (i.e., all the streams have potentially different different methods). For stream 1, we have:

$$Qi^{[1]} = Qo^{[source]} \tag{5.8}$$

$$Cai^{[1]} = Ca^{[source]} \tag{5.9}$$

For stream 2:

$$Qi^{[2]} = Qo^{[1]} \tag{5.10}$$

$$Cai^{[2]} = Ca^{[1]} \tag{5.11}$$

For stream 3:

$$Qi^{[sink]} = Qo^{[2]} \tag{5.12}$$

$$Cai^{[sink]} = Ca^{[2]} \tag{5.13}$$

The streams possess all the necessary information to relate the adjoining units. Whereas the units possess an internal state, streams have the important mission of storing the structure of the process. Zobel and Lee (1992) describe some problems when interconnecting units in a similar system, due to dimensional incompatibility or different scaling constants. They suggest two methods to solve this problem: either to provide for outputs and inputs of more than one type for each object, or to allow the user the facility for modifying the object to achieve compatibility. The approach taken in this work overcomes these consistency problems. The exact description of every input and output is known by the stream, which transforms outputs into the expected type of inputs. There are several stream classes; the choice of which stream object is created at run-time depends on the class of the units it connects. The user need not be aware of this distinction since the choice is automatic. Because they store the structure of the process, streams implement, in practice, relationships (which function as constraints) between the instance variables of the unit model objects. These constraints are generated at run-time and depend on the configuration of the process model that has been selected by the user. Since the streams are the only objects that possess this information, they can be used to make the view objects consistent with these constraints (e.g., disable editing of a panel item if the value it displays is determined by some constraint and can no longer be freely changed). If the system is being applied to dynamic simulation, and the process is continuous in time, a sampling period short enough to follow the process dynamics must be selected. All the units update their state independently for that time step, after which the relationships between them are restored by sending updating messages to the streams. As mentioned before, this is an aproximation, since all the variables are interdependent and the system should be solved simultaneously. The numerical process will tend to the simultaneous solution as the sampling interval tends to zero. This kind of decomposition lends itself immediately to concurrent calculation of the state of each unit.

A similar form of decomposition, although for a different purpose, has been used in the Stream Machine (Barth, 1986), which consists of concurrently executing modules (equivalent to units), that communicate through data streams (equivalent to streams).

5.6 Object-oriented Wrappers

A considerable amount of existing software is available in non-OOP languages. Hybrid development environments, which graft object-oriented concepts onto existing procedural languages, are necessary to maintain existing systems. Advantage can be taken of implementing computation-intensive operations in a native procedural environment and attaching them to an object-oriented system (Forde *et al.*, 1990; Kamath *et al.*, 1993). In this case, existing software can be "wrapped" inside a purpose-designed, service-based interface (Coad, 1991). In practice, Fortran77 still emerges as one of the languages most used in numerically-intensive scientific simulation, sometimes in conjunction with other languages for the graphical user interface (Bār and Zeitz, 1990; Shaw, 1992). This is mostly due to the wide number of specialized routines of numerical methods available.

In this study, C++ classes were developed for the objects, and all the relevant data are stored in the object's instance variables. However, calls to functions or routines in another language (in this case, Fortran77) may be wrapped by updating method(s) attached to classes. The fact that methods may call procedures written in different languages does not interfere with the concept of data abstraction since the external interface of the object is not affected.

The interactive application is structured according to object-oriented concepts, and Fortran77 is used only in localized and encapsulated sections of the program, i.e., the computationally-intensive operations.

5.7 MVC-based Framework Developed for the Prototype

In the previous section, the decomposition of the domain-specific component into objects, as used in this work, has been described. However, in order to develop a structured interactive application, this component must now be integrated with the graphical user interface components. As mentioned before, a solution to this problem is the definition and implementation of a suitable *framework*.

Smalltalk's Model-View-Controller MVC (Goldberg and Robson, 1983) paradigm is ideally suited for the development of this kind of application. The *model* objects map directly to the components of the physical system, whereas the *view* objects represent the type of display desired for the output of the models and the *controller* objects enable the user to interact with the models or the views.

Besides the fact that Smalltalk is not likely to provide fast execution, windowing systems like X Window provide more flexibility (e.g., running the simulation programs either on a mainframe or spread through the network, and displaying the results visually on a workstation, X Window-based PC or X terminal), allowing full advantage to be taken of heterogeneous networks of computers.

In this work, a modified version of the MVC paradigm has been developed in the form of C++ classes and is described next. The objects used in the modified MVC framework are at a higher level than the objects in Smalltalk and will be described in greater detail in Chapter 6. The selection of a framework — and therefore the selection of an object-oriented paradigm — is a major design decision in which, by analysis of the functions that must be performed by the application, a suitable behavioural composition, and therefore the main groups of classes needed, is chosen. Although the classical waterfall model (see Chapter 2) can be used to describe the software development so far, it becomes inadequate as soon as the design of each inheritance tree and individual classes starts due to the fact it does not provide for iteration. Chapter 6 will further refer to the actual design of the classes for each inheritance tree.

5.7.1 Modified MVC Paradigm

The application is basically composed of model, view and controller objects. Most additional objects (either widgets provided by the toolkit or smaller, auxiliary objects like graphical items) are attached to one of the above.

In this context, *models* are sets of data and behaviour functions (*methods*) that are intended to simulate physical entities. Models can be *units* or *streams* (if related to the domain component), or objects that manage the structure of the application.

All the visible elements such as drawing areas, menus, panels, buttons, etc. (toolkit widgets) belong to a *view*. A *view* is an entity which displays, totally or partially, the state of the system. Because user input is received by the displayed objects, i.e., the currently mapped windows, *views* receive and recognize physical events.

Finally, because *controllers* control user input, a *view* immediately redirects user input to its *controller*, executing its appropriate method. *Controllers* are therefore, basically, collections of user input handling methods.

Since the overall system is decomposed into smaller components, i.e., several *models*, the existence of supervisory objects is necessary to coordinate the interactions between them. In MVC terminology, that corresponds to the existence of a *super triad*, i.e., one *super model*, one *super view* and one *super controller*.

The super_model manages the interactions in the sense that, among other data and functions, it keeps an updated linked list of all the current models and sends updating messages to all of them. If a model is deleted so are all its views and associated controllers. Note that the actual structure of the process is not stored by the super_model.

As in Smalltalk's MVC, each view knows its controller and its model; each controller knows its view and its model. However, in this extended version of MVC, each model knows its view as well, although its knows nothing about its type. This makes communication between the model and the view (for example when updating the view) *direct*, instead of indirect as in Smalltalk.

Although each model may have more than one view, it is associated with a *primary view-controller* pair only. This is defined in the Open_View method of the View class, where the relationships between the elements of the triad are set. The dependency mechanism has been implemented in a different way from Smalltalk's. When a new view is created and assigned to a model, the view instance of the model is checked. If it has not been set yet, the view is set as the model's primary view; if it has, then the new view will be added at the end of the series of views for that model.

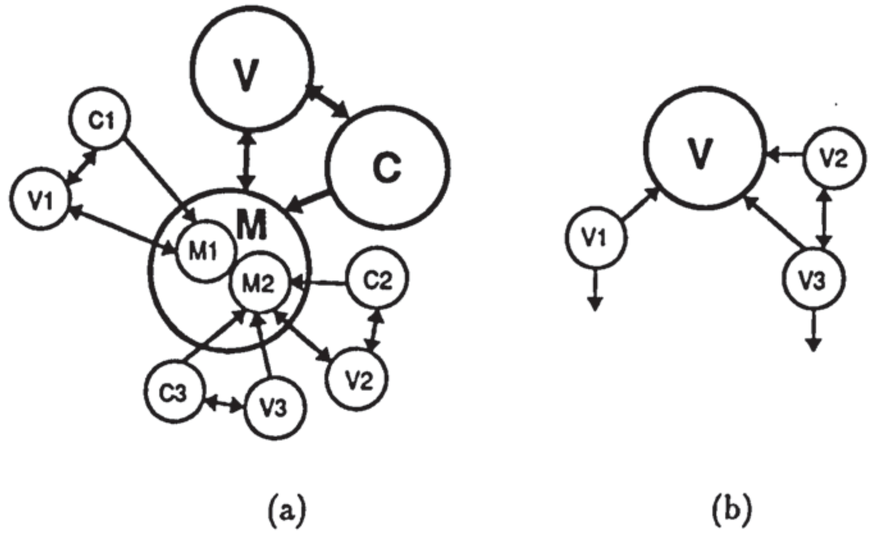


Figure 5.4: Extended MVC-based framework.
(a) Organization of the MVC triads. (b) The views as the navigational elements. M = Super Model, V = Super View, C = Super Controller. M contains a linked list of all the models (M1 and M2). V1 = Primary view of model M1, V2 = Primary view of model M2, V3 = Secondary view of model M2. Note that M2 does not know V3. (a); an *update* message would first be sent to V2 and then, in case V2 pointed to a secondary view, to V3 (b).

Each view has a pointer to a dependent view, which may or may not have been set; if it points to something, then at least one secondary view exists for the same model. The *update* message propagates through all the dependent views until one is found which leads to no further views.

Each view also has a pointer to its previous view, as well as the already mentioned following view, allowing for navigation along the views that have the same model. Each view has a pointer to the *super_view* of the system, which in turn knows the *super_model*, making it possible to access views and controllers corresponding to any of the currently existing models.

Fig.5.4 shows the structure of the MVC triads. The super_model contains all the global data necessary to the simulation (in this case, the numerical parameters for the integration) and has views of its own as well. It keeps track of time, i.e., the *clock* is associated with the super_model. Since this type of decomposition of the overall model into submodels results, as stated above, in an approximate solution, another object which may in the future be associated with the super_model is an *accuracy checker*, which can check if the currently employed time step after which the interactions between the objects are reset is suitable or must be decreased. Note that this time step is not necessarily the same used in the integration of the differential equations that describe the state of the units (the integration step may be considerably smaller, if appropriate, and different for each unit). In this version of the implementation, the super_view is a non-visible, non-interactive view, and the super_controller is a dummy controller. The super_view is however used as the *owner* (using XView terminology, its frame is the *parent* frame of the other frames) of all the other views. Since the super_view is destroyed only when the application exits, this guarantees that any currently existing window can be dismissed and later retrieved. Further mention to the *parenthood* relationship will be made in Chapter 6. For the sake of efficiency, the super_view may in future implementations perform other functions as well (such as keeping a list of all the views).

5.8 Distributed Structure of the Prototype

As mentioned before, the hardest problem found in simulation is usually providing adequate response time; this factor becomes critical in interactive software. The ideal situation occurs when an arbitrary number of parallel processors executes simultaneously the updating procedures for each object, similarly to what happens in the real world (Rumbaugh, 1991).

In this study, two different distribution approaches were implemented. In the current version of the prototype, PVM is used to distribute concurrently the computationally-intensive methods throughout the network. This approach blends well with the object-oriented paradigm and resulted from an analysis of the main deficiencies of the distribution scheme used in the previous version of the prototype, based on RPC. An overview of both approaches is made in this section.

5.8.1 PVM-based Distribution Approach

XWindow provides already a certain degree of distribution, since the machine where the application is executed does not have to be the same one where the results are displayed. However, in order to decrease response time, the application itself must be able to start multiple concurrent threads of control.

Such a facility was implemented in this study using the PVM (Parallel Virtual Machine) software.

Parallel processing can be implemented at several levels, namely at the object level (in which objects are allowed to reside in different nodes) or at the algorithm level (where a method uses a parallel algorithm). As mentioned in Chapter 4, the feasibility of a distributed object-oriented approach in a distributed environment is subject to discussion (Carré and Cléré, 1989). On the other hand, the development of parallel versions of numerical algorithms is out of the scope of this work. A third alternative consists of the development of a master program, written in an object-oriented language, together with the distribution of methods throughout the network, which may be an effective way of taking the best advantage of both existing computational resources and software.

In the prototype developed, each object can make its computations locally or, alternatively, start a PVM process which can be executed on another machine (among those which are currently part of the virtual machine). This decision must be made by the programmer based on load balance considerations but, in general, a fully distributed approach will lead to lower response times. Furthermore, because PVM allows calls to be made from either C (or C++) programs (used in the case of distributing the methods throughout the network) or Fortran (used in the case of implementing distributed versions of the algorithms), parallelism can be implemented at any level. The spawned processes themselves may be written in an OOP or procedural language, mostly depending on the available software. Because PVM enables concurrency, all the updating methods can be executed concurrently. The pattern used in this work is a typical master-slave paradigm in which the GUI module is the master and the created processes, which execute the numerically-intensive calculations, are the slaves. Synchronisation is ensured since PVM provides asynchronous blocking send and asynchronous blocking receive functions. A blocking receive function returns only when the data is in the receive buffer. Before starting the calculations, each slave waits until the data it needs are available. The master module then waits for messages from all the slaves, after their computations are done, before resuming execution (see Fig.5.5). Afterwards, all the slave processes wait for another message from the master before continuing the calculations. This has the advantage of keeping the simulation processes in stand-by, therefore preserving the state of all the previous calculations, which would not happen in the case of a distribution strategy such as the use of RPC (unless all the variables involved were declared *static* or similar).

This strategy consists of the implementation of distributed methods, i.e, the spreading of computationally-intensive methods, or parts of methods, throughout the network, to reduce execution time; it does not aim to become the implementation of a distributed object-oriented language. Standard software (languages and compilers) is used. The model classes include methods to create, send messages to resume calculations, receive messages after calculations are done, and kill or send other UNIX signals to PVM processes running on the network. The remote processes can also send UNIX signals to the interface. Both the interface and the processes are provided with signal handlers, which will be described in greater detail in Chapter 6. Fig.5.5 shows a diagram of the distributed structure of the prototype.

If a model object chooses to use a PVM process for its calculations, the updating process must be divided into two stages. Resuming the execution (the process must have already been created) is performed by the Start_Update method, whereas the End_Update method waits for a message from the remote process after completion of the calculations. Otherwise, the updating procedure is performed in the Update method and the default Start_Update and End_Update methods are not overridden (do nothing). The order of the updating messages sent by the super_model must be the one shown, so that the local computations are executed simultaneously with the remote ones and do not hamper the process. Other methods of the model classes necessary to the management of the distributed structure include Create_Process, End_Process and Send_Signal.

5.8.2 RPC-based Distribution Approach

The prototype was initially designed as a graphical user interface to facilitate the interaction with a specific set of programs, namely for the dynamic simulation of the chemical recovery cycle of a paper pulp mill (Pais *et al.*, 1994).

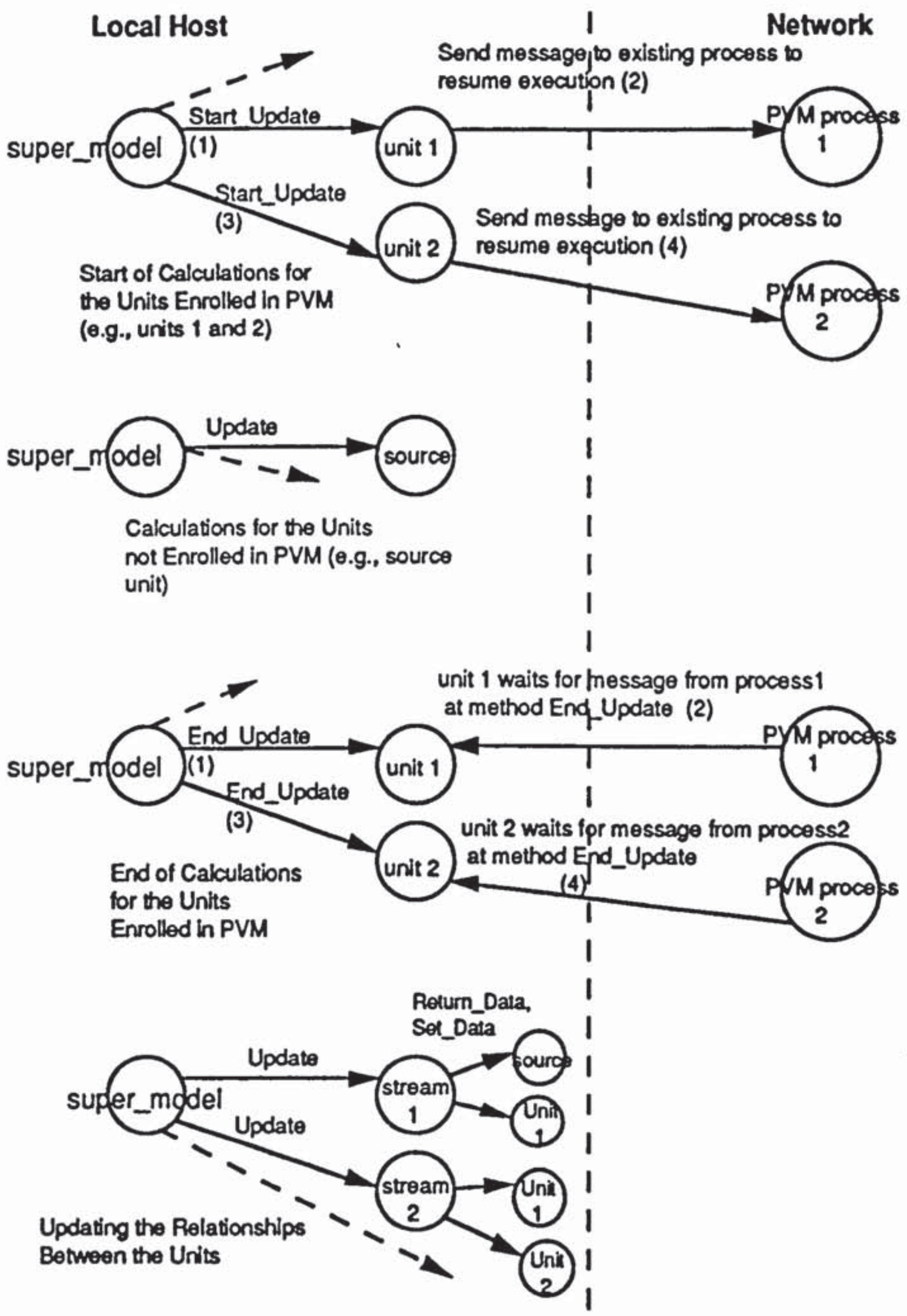

Local Host
Network
Send message to existing process to resume execution (2)
super_model
Start_Update (1)
unit 1
PVM process 1
Start_Update (3)
unit 2
Send message to existing process to resume execution (4)
PVM process 2
Start of Calculations for the Units Enrolled in PVM (e.g., units 1 and 2)
super_model
Update
source
Calculations for the Units not Enrolled in PVM (e.g., source unit)
unit 1 waits for message from process1 at method End_Update (2)
super_model
End_Update (1)
unit 1
PVM process 1
End_Update (3)
unit 2
unit 2 waits for message from process2 at method End_Update (4)
PVM process 2
End of Calculations for the Units Enrolled in PVM
Return_Data, Set_Data
super_model
Update
Update
stream 1
source
Unit 1
stream 2
Unit 1
Unit 2
Updating the Relationships Between the Units

Figure 5.5: Simplified distributed structure of the prototype. (See also Fig.5.3).

The programming language used was C, which enabled the use of the selected windowing system (X Window) and its toolkits. Since the configuration of the physical system was known, the simulation routines (written in Fortran77) were all called from a module also written in Fortran77. This implied that the integration process was performed simultaneously for all the components of the physical system, i.e., there was only one simulation program. This approach soon revealed itself to be highly inefficient. In the first place, the structure of the interface became, due to the absence of a framework, a large agglomerate of intricately related widgets. In the second place, the fact that minor changes in the physical system involved code re-writing and re-compiling was an overpowering disadvantage. Even if the physical system of interest is basically the same, the software used must allow minor changes to be made on-line. It is not practical to edit the source code files and re-compile whenever, for example, an extra valve is added to the current configuration. None of these disadvantages are present in the latest, object-oriented version of the prototype.

Still, the first version provided the possibility of executing the simulation program in a machine other than the one where the application was executed, and, furthermore, "freeing" the application (e.g., for analysis of results obtained so far) in the meantime, so that the interface and the simulation processes were executed concurrently. Communication of the interface with the remote simulation program was done in two ways, namely by means of normal RPC requests, and via flags written to log files.

Sequential RPC Calls

Whenever the response time of a request was short enough to be acceptable for on-line purposes (e.g., when initializing the simulation data), a sequential RPC call was made and direct transfer of parameters was employed. The RPC package was used to link the local C interface application to a remote C procedure which in turn called the remote Fortran77 simulation routine. A C/Fortran77 interface was developed for this purpose (see Fig.5.6). Current parameters were transferred to and from the remote C routine using the UDP transfer mechanism. This made it necessary to develop XDR (eXternal Data Representation)

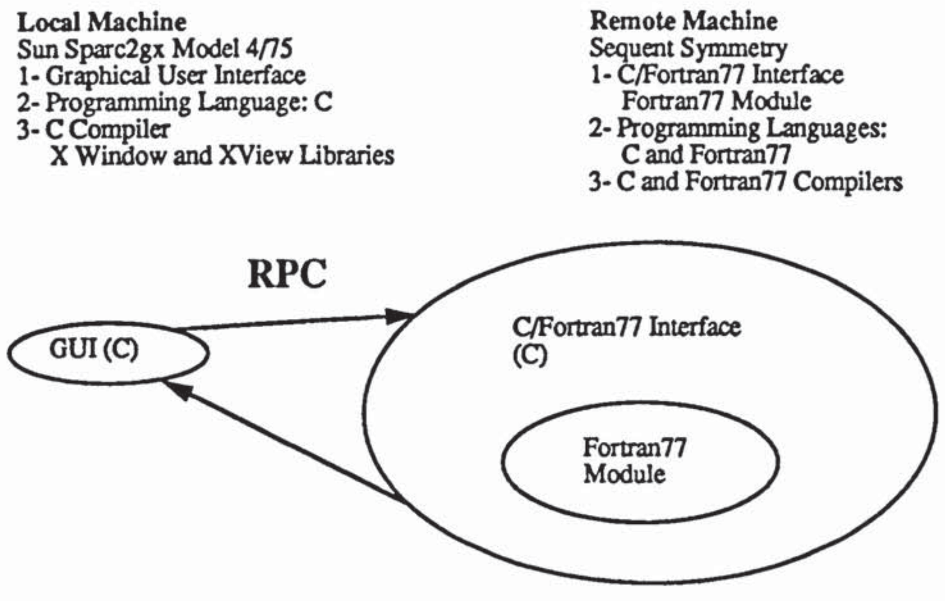


Figure 5.6: C/Fortran77 structure in the first version of the prototype.

routines which transform data particular to a specific machine into a machine-independent format.

Signal Handling Routines

Initialization (reading all the default parameters) was done via an RPC call that would normally have a successful return; sending the simulation parameters to the simulation program was also done via an RPC request. However, execution time of the simulation program was normally quite long and impredictable, since it depended to a great extent on the particular combination of parameters used for that run. In order to avoid blocking the interface until the remote routine returned, a short timeout for the RPC call was specified. An unsuccessful return occured and was ignored, while execution continued in the remote machine. To compensate for the loss of the initial client/server link, flags were written to a log file whenever a change occurred in the execution status of the simulation program (see Fig.5.7). This change could be the availability of new results or events such as abnormal termination (e.g., due to arithmetic exceptions) of the simulation programs. This led to the installation of a series of system signal handling routines in the simulation programs, which caught signals generated by the system and wrote the appropriate flag to the log file.

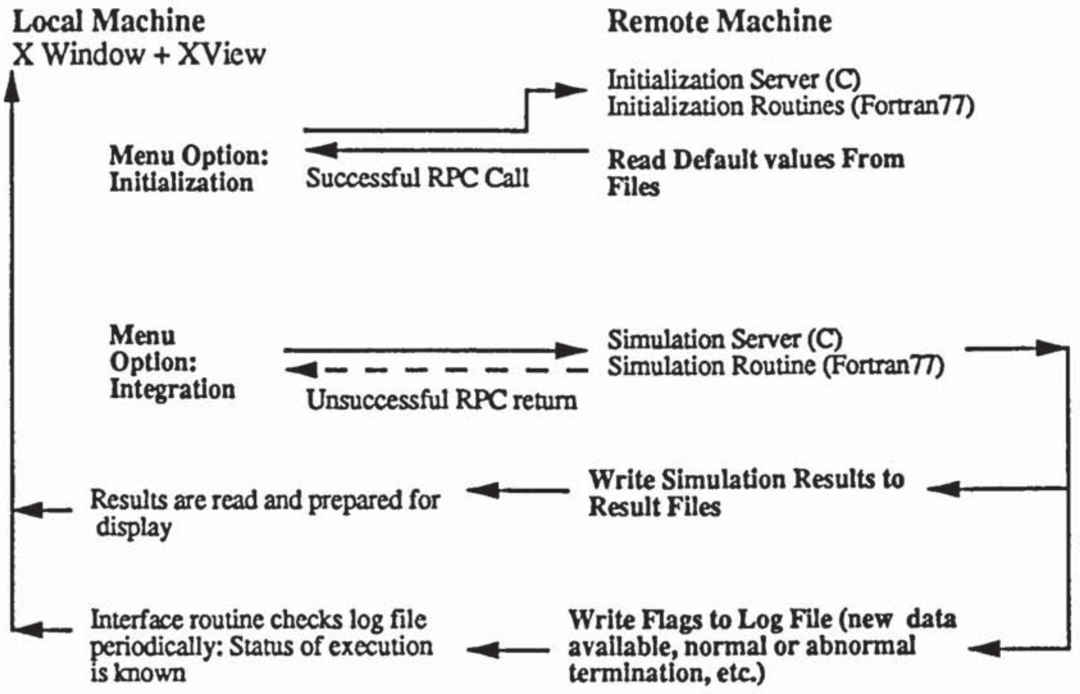

Local Machine
X Window + XView
Remote Machine
Initialization Server (C)
Initialization Routines (Fortran77)
Menu Option: Initialization
Successful RPC Call
Read Default values From Files
Menu Option: Integration
Simulation Server (C)
Simulation Routine (Fortran77)
Unsuccessful RPC return
Results are read and prepared for display
Write Simulation Results to Result Files
Interface routine checks log file periodically: Status of execution is known
Write Flags to Log File (new data available, normal or abnormal termination, etc.)

Figure 5.7: Communication between the interface and the remote simulation program in the first version of the prototype.

The interface possessed a routine that periodically checked the log file and took apropriate action according to the flag value, such as displaying a dialog box to alert the user to the crash of the simulation programs. This methodology made it possible to know what was happening to the simulation programs while the interface remained free for the analysis of results produced so far. System signals could be generated either by the system — and intercepted and handled by the signal handling routines — or by the user (e.g., to stop, pause or resume the integration) and sent to the remote program.

This approach was a way to get around the basically sequential functionality provided by RPC. Note however that communication via data written to files, besides being inefficient in the sense that it takes time to open a file, read data and close the file, was also dependent on the fact that both the local and remote machines shared the same virtual file store, with common pathnames, and therefore all the files used were transparently available to both processes. It would be far more complex to implement if the local and remote machines had separate file spaces.

In conclusion, procedural languages such as Fortran77, albeit suited to numerically-intensive calculations, are inadequate for the on-line definition of the process model. Languages like C++ provide both efficiency of execution and object-oriented features which enable the utilization of an object-oriented framework. The application can then be systematically factored into the components of the framework, which include objects related to both the problem domain and the graphical user interface. In this work, an extension of the MVC framework was developed which provides classes for the management of the simulation. Each object may create a separate process which performs concurrently the calculations associated with the updating methods.

So far, the foundations used in the development of the prototype have been described; Chapters 6 and 7 provide a more detailed insight.

Chapter 6

Standard Class Library of the Prototype

6.1 Introduction

The user of a simulation package should not be expected to be a simulation expert or even familiar with the technical simulation details and terminology (Treu, 1988). Simulation programs are often poorly understood by the user, which results in the inability to use them to their full potential (Wright *et al.*, 1990). Though modelling and simulation have become an important part of most engineering disciplines, many well-known and accepted programs have become extremely large and complex, precluding use by inexperienced professionals.

The complementation of simulation programs with a graphical user interface, which provides graphical representations of the physical system itself and of its state, and specifically indicates how to perform the activities associated with the simulation, makes ít significantly easier for users to take full advantage of simulation packages. The interactive interface must be flexible enough to let the user manipulate the system model and the specified parameters, preferably using graphical tools. After the results of the simulation are returned to the interface for user viewing, thus providing visual feedback, the user must be allowed to change some of the specified parameter values or constraints or even return to modifying the configuration of the system model itself.

The prototype UIMS (User Interface Management System) developed in this work aims to offer a structured framework on which to build graphical user interfaces for interactive simulation, providing a tool for the on-line definition of the configuration of the physical system and management of the corresponding simulation. Rather than presenting an interface for interaction with a specific simulation model (or even problem domain), it offers a means by which user-written simulation programs (not necessarily written in an object-oriented language) can be imported into an object-oriented, interactive graphical environment with minimum effort. It takes full advantage of the widget libraries provided by toolkits (in this case, XView), grouping widgets into higher-level *view* objects, which make it easy to create complex displays, such as interactive line charts, at run-time. Although it is written in C++, which does not directly support concurrency, it uses the PVM libraries to enable the creation and management of concurrent processes, not necessarily on the same host machine but rather on a desired set of hosts available in the network. These functions are encapsulated by object-oriented methods so that the complexities of concurrency management are hidden from the user.

As mentioned in Chapter 5, C++ was the language selected for the prototype, together with the windowing system X Window and the XView toolkit. This ensures a high level of portability since only standard and widely available software is used. In Chapter 5, the framework and the type of distribution developed for the prototype were described. The current chapter describes in greater detail the class hierarchies created and the methodology used to merge this layer of software with the underlying X Window and XView layers. The C++ base classes, or *standard* classes, follow a pre-defined structure and include features specifically dedicated to interactive simulation (including concurrency, as mentioned before). Please note that the term *standard* is applied, in this and the following chapters, to the domain-independent class libraries developed *in this work*, not to the (very few) class libraries provided by plain C++. Views that offer common types of graphical representation, such as line and bar charts, are also provided. User-defined classes can be sub-classed from the standard classes provided by the prototype in order to enable the application of the UIMS to any problem domain area. This chapter thus describes the design principles used in the prototype, together with some relevant implementation details. Furthermore, it describes the graphical tool developed to enable the user to define the system model on-line. The prototype developed has been tested by application, together with simulation routines developed in Fortran77, to the generation of a graphical user interface for dynamic simulation of a chemical process system; further details of this case study are given in Chapter 7. Due to the extent of the source code written during this work, only a list of the files used by the prototype is presented in Appendix E, including the class protocols (*.h) and implementations (*.C). The domain-specific classes necessary for the application of the prototype to the case study, which will be presented in Chapter 7, are also listed in Appendix E. A tar file (XProc.tar) containing the source code itself is available via ftp (see Appendix E, section E.1).

6.2 Classes Provided by the Prototype

The working prototype is composed of interacting objects, most of which can be created and deleted at run-time. The framework itself is composed, as described in Chapter 5, of *model*, *view* and *controller* objects. Although the basic "triad" element is formed by the same components as in Smalltalk, the implementation of each component is totally different, as well as some of the functions it performs, i.e., no attempt was made to reproduce Smalltalk's class libraries.

In addition to the three main types of classes (models, views and controllers), three other inheritance trees were found to be necessary, namely model-view adaptors (which enable the view classes to be independent from the model classes), graphical items (which enable the graphical definition of the process model) and exception handlers (which enable detection of abnormal situations and increase the robustness of the program). The design of each of these trees proceeded in an iterative pattern, where sometimes the addition of a new derived class made it necessary to change its superclasses. This phenomenon is especially important in C++, where the base classes must have declared as virtual all the methods that may be later required by the derived classes. Each inheritance tree, or parts of it, can be thought of as a *cluster*, or set of related classes, as defined in Chapter 2; the development of this software followed, basically, the cluster model. A difference lies however in the fact that simplified implementations of each cluster were assembled and tested together, several times, before any of

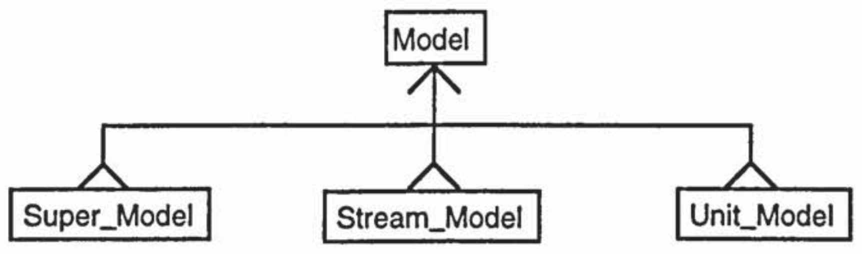


Figure 6.1: Standard model class hierarchy.

them reached its final stage. This enabled an evaluation of the performance of the system as a whole at early stages of the cluster life cycle and resulted in input to the next analysis/design/implementation stages of most clusters.

In this section, a description of the class protocol for the standard classes of each inheritance tree is made.

6.2.1 Models

The *model* objects, which intend to simulate physical entities, are essentially domain-specific; this implies that the standard classes contain a very limited number of model classes. If the UIMS is to be applied to a specific problem or set of problems, then libraries of more specialized classes can be created which will reduce the work and amount of code involved in creating the classes needed by the application. For example, if the UIMS is to generate interfaces associated with the simulation of chemical processes, then the library of model objects could include the Valve, Pump, Reactor, etc. base classes. In the present stage, the model classes provided by the UIMS (domain-specific model classes are described in Chapter 7) are presented in Fig.6.1. The notation used is that proposed by Rumbaugh *et al.* (1991), where each class is described by a rectangle with the name of the class in it. The rectangle can be divided into several regions, where the top region is reserved for the name of the class, the middle region for its instance variables and the lower region for its public methods. Some of these regions may be omitted as appropriate. As mentioned above, in order to fully use the inheritance and polymorphism capabilities provided by C++, the base classes must contain, declared as virtual, all the methods which are likely to be overridden by the derived classes. The definition of the protocol of the Model base class is therefore not trivial (see Fig.6.2.). Coplien (1992) warns against the encapsulation of instance variables only to provide means (methods) to access and change them. However, in this specific case, the objective of the interface is exactly, among other things, to access and modify the parameters used in the simulation, i.e., the data of the models; this is performed by the controller objects, which manage the interaction with the user. This led to the need to provide for, in the base class, public methods to access and alter the data of the model. This approach minimizes the possibility of accidental corruption and was thought preferable to declaring the data themselves public. Furthermore, the model subclasses can, in the re-definition of these methods, prevent alteration of certain instance data and issue a warning message (or return an error value) if this is attempted. Because the data specific to each model is not known, and must be unrestricted, generic functions like Return_Double_Data are provided, which take a string as a parameter and, depending on its value, return a specific datum or an error value (if the string is not recognized by the model). Although this is an implementation detail, the definition of generic methods to handle the data of the models is necessary in a system where the new model objects that are created at run-time can belong to any class and have any type and number of instance data. The method Initialize refers, as it indicates, to the initialization of the model (usually reading data from a file); the following group of methods refers to the management of the simulation, either concurrently or non-concurrently (as described in Chapter 5), Write writes the results to files, and Reset resets the state of the model to its initial state. The next group of methods is dedicated to the management of the system model and was included specifically for use by the super_model (see Fig.6.3). As mentioned in Chapter 5, the super_model contains a linked list of all the models that compose the currently defined system model. Methods First_Model_List and Next_Model_List provide the means to handle this linked list, returning its first element and the next element, respectively. Methods Add and Clear add a new model to the list and delete the whole list, respectively.

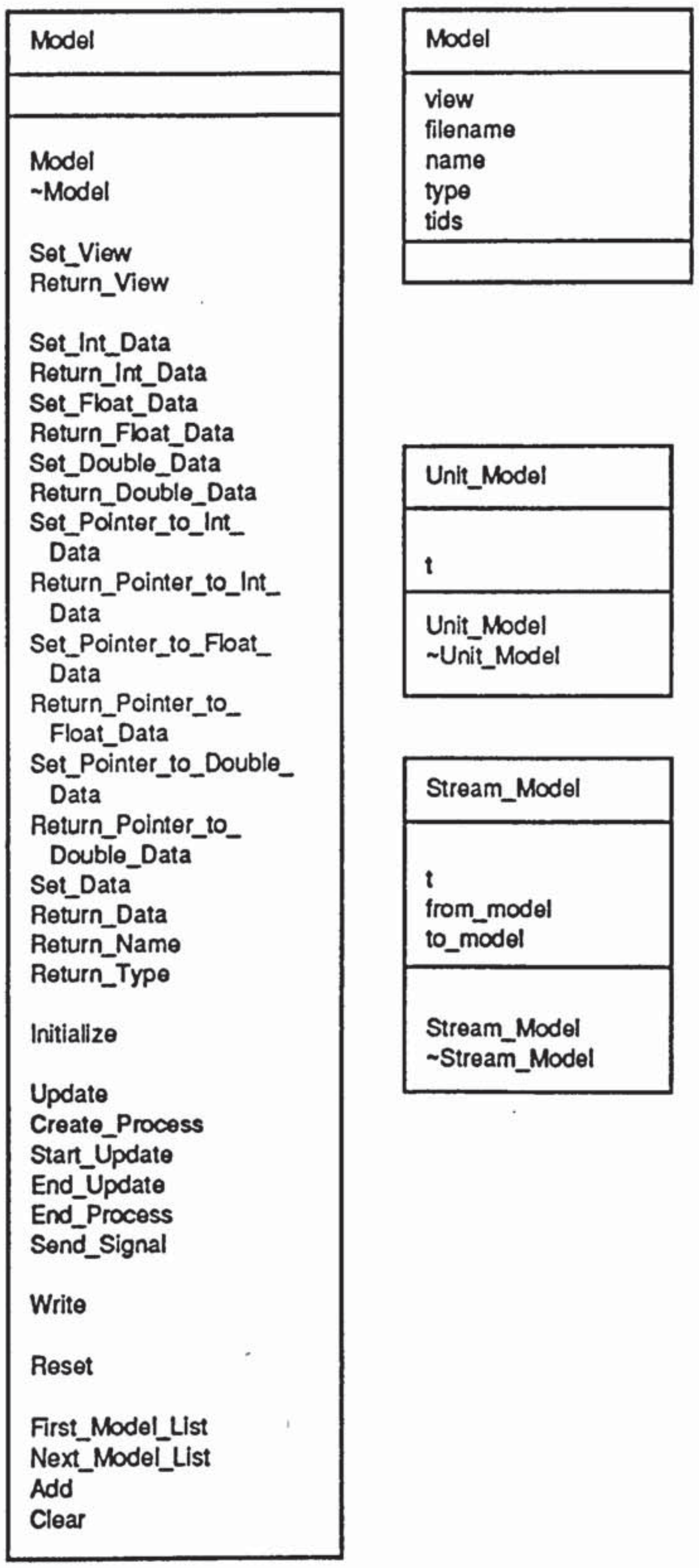


Figure 6.2: Protocol and instance variables of the standard model classes. (~ denotes the *destructor* method).

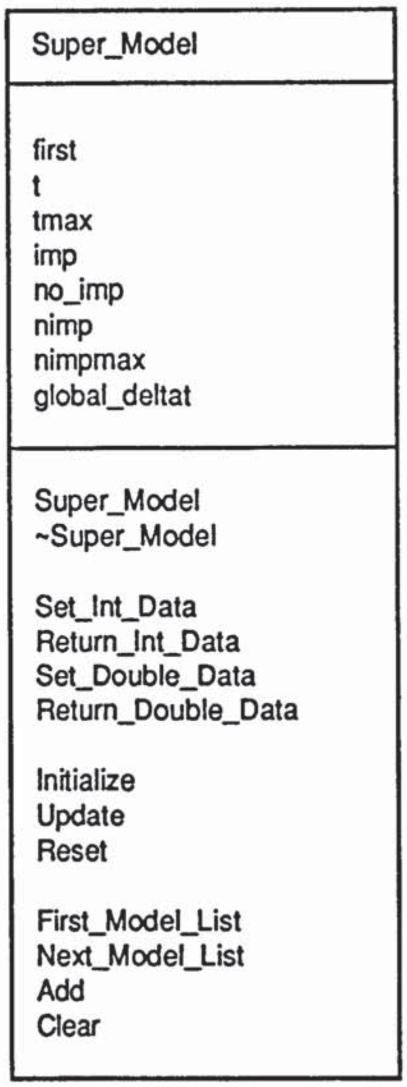


Figure 6.3: Super_Model class protocol and instance variables. (~ denotes the *destructor* method).

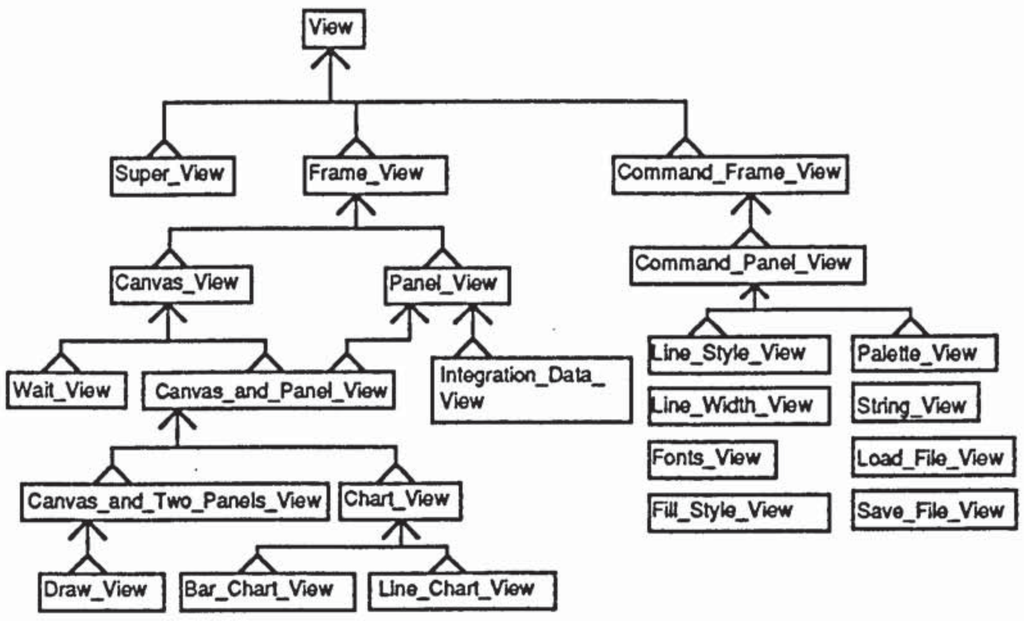


Figure 6.4: Standard view class hierarchy.

6.2.2 Views

To take direct advantage of the toolkit objects, and make it easier for the user to create views that are formed by several widgets, views are toolkit specific. In XView, a *frame* is a container for other windows (Heller, 1993). It manages the geometry and placement of subwindows such as canvases, panels, text subwindows and scrollbars. These subwindows cannot exist without a parent frame to manage them. Although frames can overlap, subwindows within a frame occupy fixed, non-overlapping positions.

Using XView terminology, a view is associated with a *frame*. A canvas is a drawing area, and a panel is a control area. Standard classes of views that make direct use of XView widgets have been created, such as the basic classes Canvas_View, Panel_View, Canvas_and_Panel_View, Command_Panel_View, etc. (see Figure 6.4). The Canvas_and_Panel_View class, for example, creates a frame, a canvas and a panel and registers the appropriate procedures as callbacks. The standard view classes provide the basis for the creation of specific types of views, but they still have to be customized by the user. For example, several panel objects may be created for each panel (buttons, sliders, etc.). Therefore, the standard view classes contain empty methods for, for example, the generation of the corresponding panels. The user creates new subclasses of the standard ones where these methods are over-ridden. If the user wishes to create a view with a canvas and a panel, he can subclass his view from the Canvas_and_Panel_View class. New standard classes can, at any point, be added to the library. For example, it would be straightforward to create a Canvas_and_Three_Panels_View standard class, which would inherit from a Canvas_and_Two_Panels_View class. This approach is somewhat similar to the one proposed by Szekely (1990) and can be viewed as adding "a layer of software on top of traditional user interface toolkits" (sic). In this specific case, this is necessary because the types of views desired for the output of the models may be complex and include several widgets. The definition of views as high-level objects makes it possible to keep related widgets together. When a new view is created, all the widgets that compose it are created with it. Fig.6.5 shows the protocol and instance variables of the standard View classes.

The first set of methods is similar to the Model class. Whereas these methods are necessary, in the previous case, to access and change the data of the models, they are also necessary in this case since some of the views, such as the Line_Chart_View and Bar_Chart_View views, are highly interactive. These views present a range of options to the user such as colour of all the elements, text font size, line width, colour and style (for the line chart) and bar colour and fill pattern (for the bar chart) (see Fig.6.6). This implies that the user is allowed to modify fonts, colour, line styles, etc., interactively; once again, the controller objects must be able to access and change the data of the respective view. The number and type of data are not restricted in any way and so generic access methods must be provided.

The Update, Show and Hide methods update the view when the state of the model changes, and display or hide the frame, respectively. The last two encapsulate directly XView functions calls. The XView toolkit gives the programmer the possibility of attaching callbacks to widgets. For example, if the user resizes or exposes a previously obscured window (which generates WIN_RESIZE and WIN_REPAINT events), the respective event handlers are called. The prototype encapsulates this event handling mechanism into a message-driven one. When

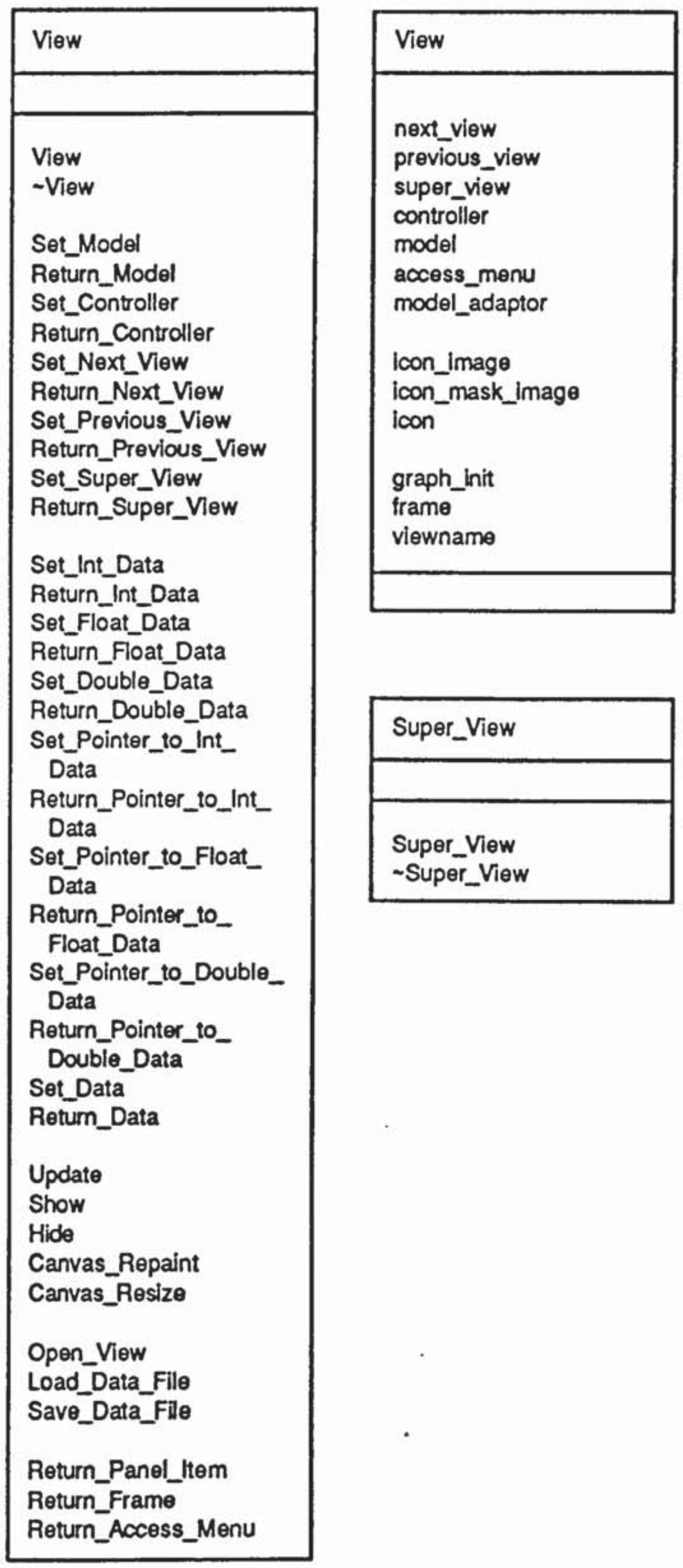


Figure 6.5: Protocol and instance variables of the standard view classes. (~ denotes the *destructor* method).

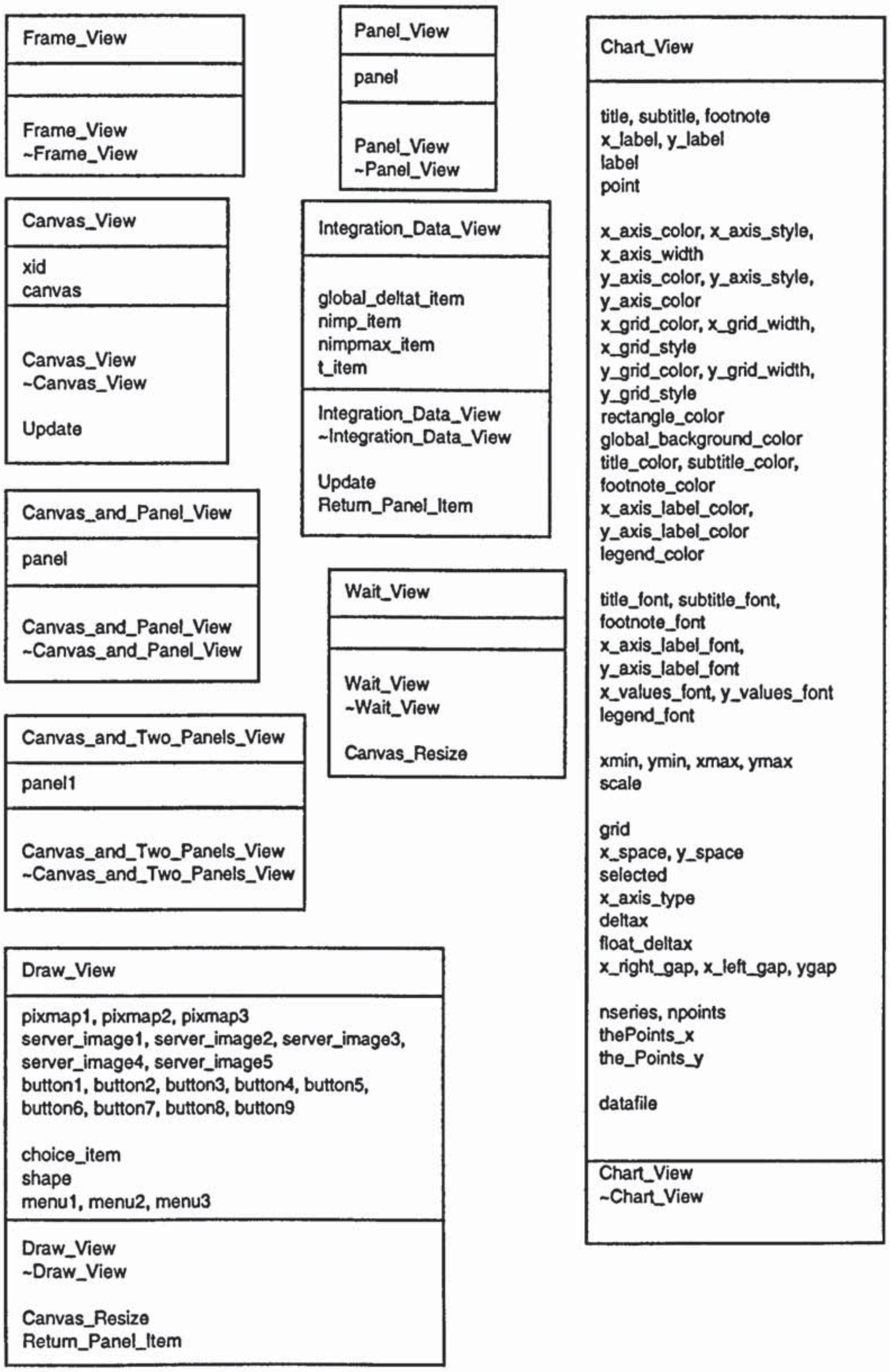


Figure 6.5 (cont.): Protocol and instance variables of the standard view classes.

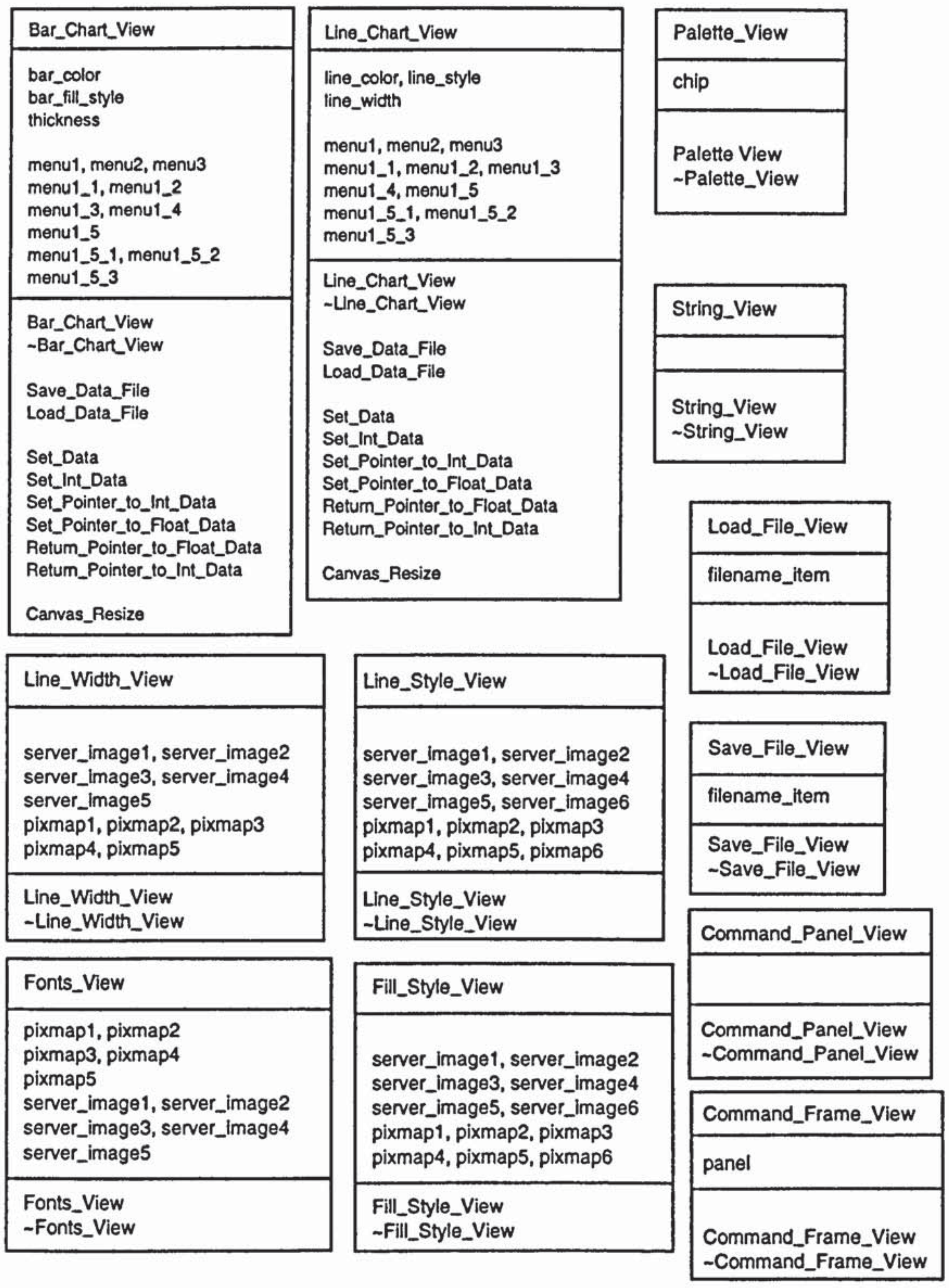


Figure 6.5 (cont.): Protocol and instance variables of the standard view classes.

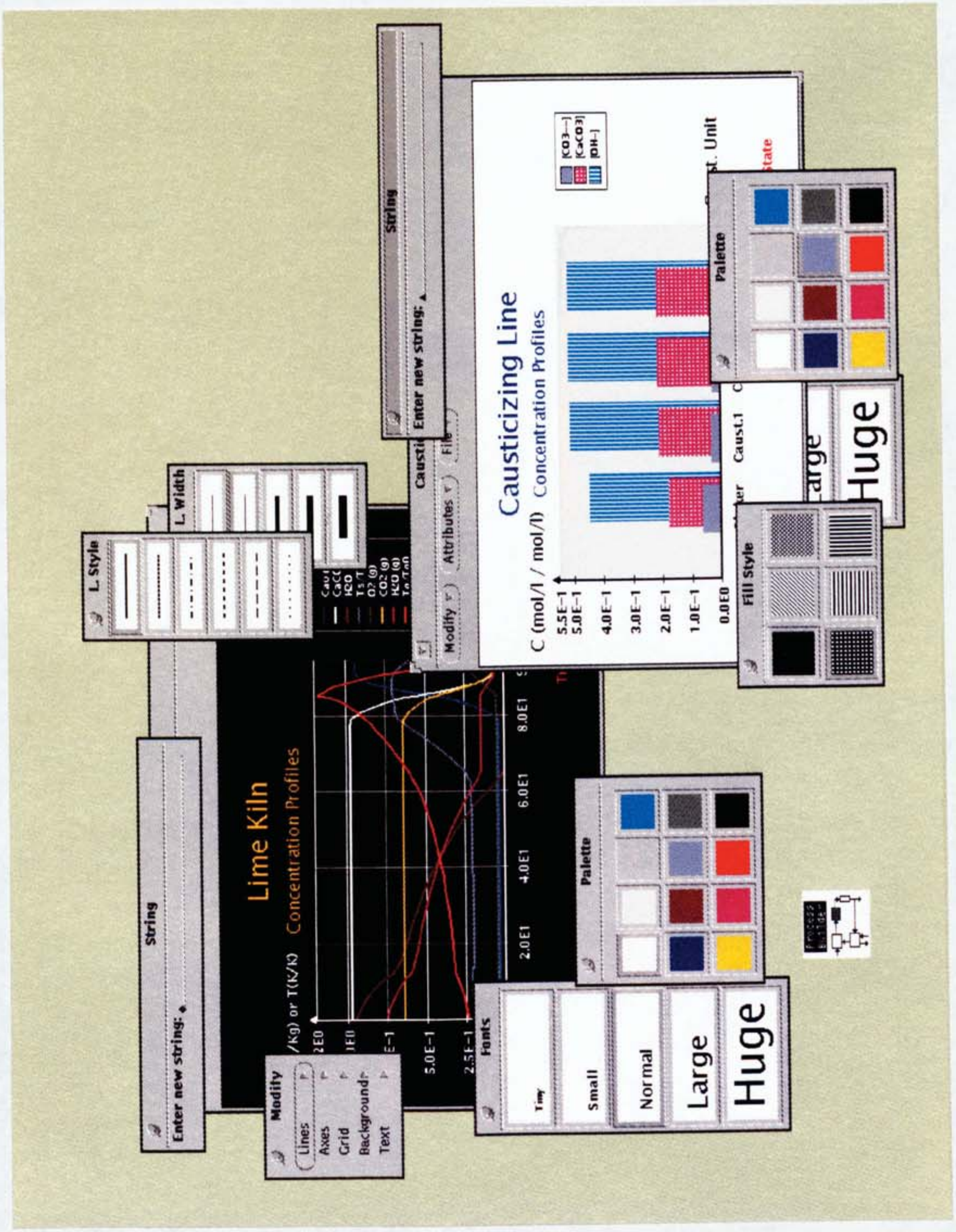


Figure 6.6: Examples of views subclassed from the Line_Chart_View and Bar_-Chart_View classes.
(Lime_Kiln_Line_Chart_View and Caust_Battery_Bar_Chart_View — see Chapter 7).

such an event is generated, a message (e.g., Canvas_Repaint) is sent to the view that contains the affected canvas, and the corresponding view method (equivalent to the event handler) is executed. This enables the event handlers to be defined as methods of a class, rather than unbound C functions. Note that not all event handlers have been associated with the views; rather, most of them have been associated with the controllers and will be described in the next subsection. These two specific event handlers have been associated with the views since they are directly related to the display characteristics of the view. The Open_View method has already been referred to in Chapter 5 and establishes the relationship between the elements of a specific triad. The Load_Data_File and Save_Data_File methods perform, as their names indicate, the retrieval and saving of view data from/to a file. Finally, the last group of Return methods enable the controller of a view to access its data, which is necessary in order to respond to certain user requests (e.g., which model parameter is changed depends on the panel item activated).

6.2.3 Controllers

Several approaches to the role of the controller objects have been described in the literature. Urlocker (1989) suggests that, in GUI's that couple graphical rendering and user interaction, the responsibility of the MVC view and control can be combined into the single view object. He maintains that the separation performed by the MVC approach (the user input directed to a view is handled by its controller, whereas the output of the model, equally directed to the view, is handled by the view itself) is difficult to learn and adds unnecessary complexity. On the other extreme, Shan (1990a, b) states that Smalltalk does not offer orthogonality, i.e., independence, between the objects that compose the framework, which severely impairs class reusability. Each controller is tightly coupled to its respective view; furthermore, the fact that the view must query the model before updating itself inserts knowledge of the application into the user interface and therefore the model, view and controller objects are in fact coupled.

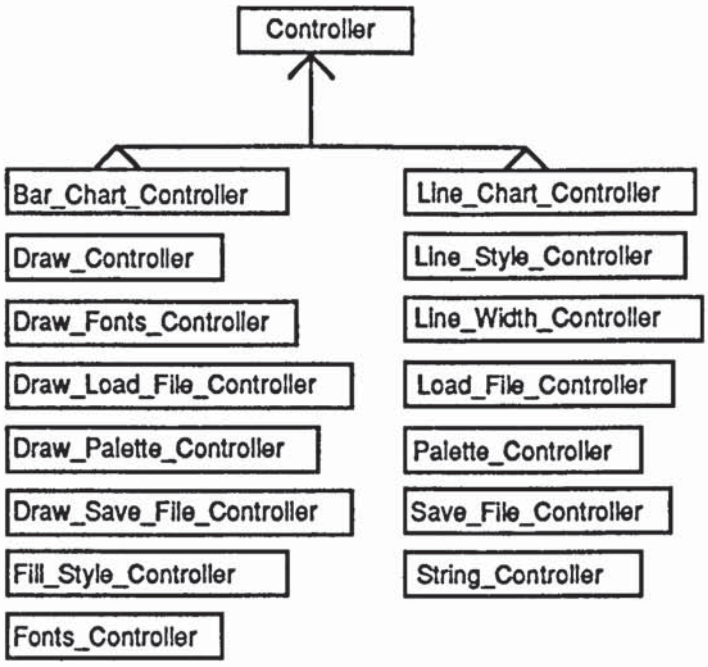


Figure 6.7: Standard controller class hierarchy.

Although it is true that creating a new view subclass, for example, will in most cases make it necessary to create a corresponding controller class, the approach used in this work for the definition of the functions performed by the controllers is similar to Smalltalk's and was thought to be the most balanced. The existence of controllers as such is beneficial since

a) the input/output management functions are distributed across two objects, thus avoiding very large and complex view classes;

b) if two views are to look the same, but respond differently to user input, only the associated controller must be different, avoiding subclassing of the views based on the way they *behave*, rather than look.

Because, in X Window and XView, event handling routines are typically associated with (take as arguments) objects that belong to the views, associating them to the controller objects leads to the need for constant transfer of data between the view and the controller. Fig.6.7 presents the standard controller classes and Fig.6.8 shows the protocol and instance variables of the standard controller classes.

Most controller classes are directly subclassed from the base class Controller. These classes are basically collections of user-input handling methods, i.e., event

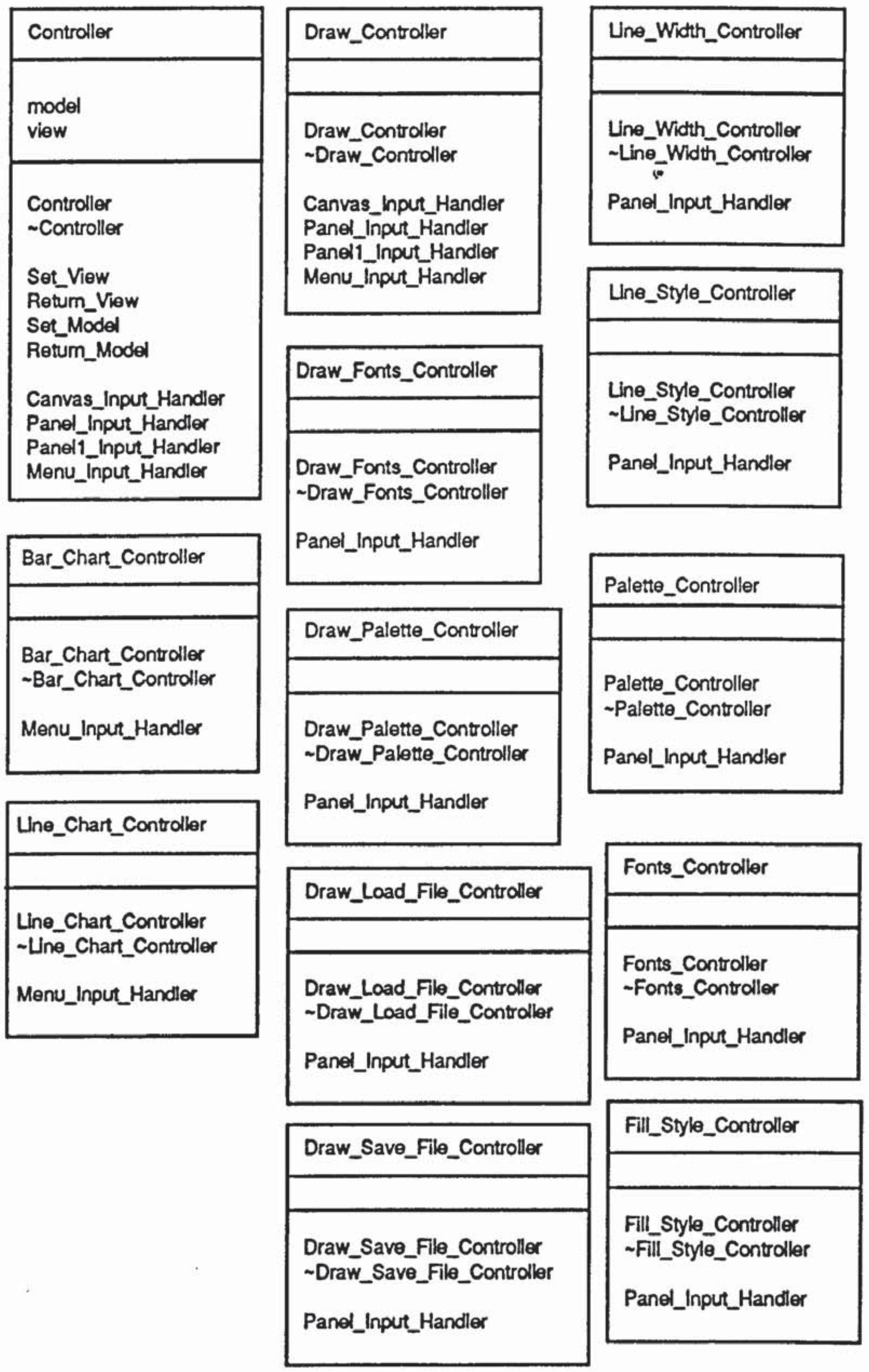


Figure 6.8: Protocol and instance variables of the standard controller classes. (~ denotes the *destructor* method).

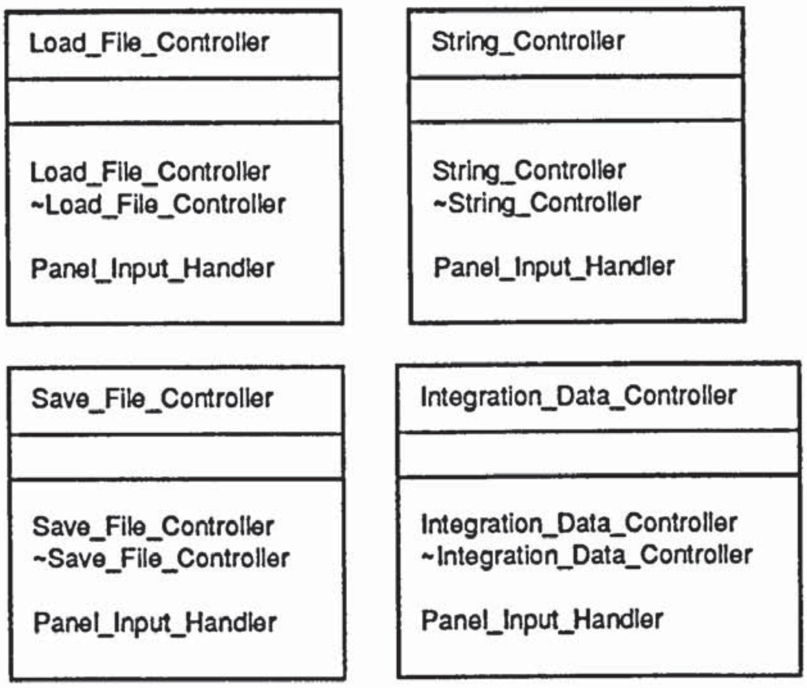


Figure 6.8 (cont.): Protocol and instance variables of the standard controller classes.

handlers which are not directly related to the appearance of the view and have been encapsulated in an object-oriented environment.

Masui (1991) points out what he considers to be a weakness of the X Window and X-based software: because the application can wait for only a single input stream, all the inputs are sent to it as "events", which have the same structure for all the inputs. This leads to the fact that event-handling routines tend to become complicated. This is quite true and led, for example, to the development of a complex Draw_Controller class, since the respective view (from the Draw_View class) receives input from both the mouse and keyboard. If several input processes can be created, each of which deals with the input from a different device, the structure of the program becomes simpler.

6.2.4 Adaptors

The existence of model-view adaptors is necessary because most models produce results which, although necessary for determining the *state* of the system, are irrevelant for graphical display. An example is the case where numerical meth-

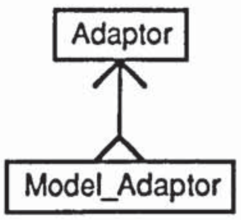


Figure 6.9: Standard adaptor class hierarchy.

ods such as orthogonal collocation in fixed finite elements are used; among the dependent variables are spatial derivatives which will not usually be included in graphical displays. In a generic case, a selection has to be made of the results in order to prepare the data for graphical display; this stage corresponds to the *graph evaluation* as described by Nielson (1991). This task must not be assigned to the model, because it must not know which kind of view(s) it is associated with; it must not be assigned to the view for the same reason. The view is an entity that, given a convenient set of data in a pre-defined format, will display it, for example as a line chart. There must therefore be an adaptor, or converter, which converts the results from the model into convenient data sets for each view. This implies that each model-view pair must have an adaptor. Adaptors can be of different types: they may convert the current data of the model into a suitable form for display, or they can store the value of a certain variable over a period in time and represent the evolution of that variable in *time*.

Fig.6.9 presents the standard adaptor class hierarchy and Fig.6.10 presents the protocol of the standard adaptor classes. The only function of the adaptor objects is to retrieve the data of the model, perform whatever operations are necessary, and set the internal data of the view accordingly.

If a view is to display results from several models, for example for comparison, an obvious strategy is to create a *composed model*, which contains pointers to the individual models. The model-view adaptor would have all the information necessary to retrieve the data from the various models and prepare them for display.

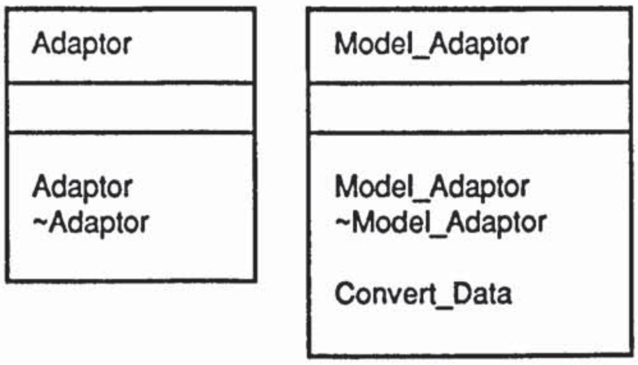


Figure 6.10: Protocol and instance variables of the standard adaptor classes. (~ denotes the *destructor* method).

6.2.5 Graphical Items

Widgets (pseudo-objects provided by the toolkit, since they are written in C) used by the prototype will not be listed here. For a complete description, please see Heller (1993).

Few toolkits pay attention to the graphical aspects of user interface programming, being specifically directed towards dialogue programming (Laffra and van den Bos, 1990). Buttons are extensions of function keys, which are accelerators for typing a command. Menus allow for the input of commands in a more friendly way. Entering lines and circles, indicating regions of interest in a work area, and direct manipulation of graphical items (and resulting semantic feedback) are examples of graphical interactions that are badly supported by existing toolkits (X toolkits included). This implied that constant use had to be made of the Xlib primitives and new classes of graphical objects had to be created as X is not a graphics system and does not offer any rule or guideline on how to handle graphics such as, for example, GKS (Peddie, 1992).

Figure 6.11 presents the C++ class hierarchies developed which, besides enabling the creation of the graphical representation of the elements that compose the physical system, also provide the basic editing operations which enable the user to define the system on-line. Fig.6.12 represents the protocol and instance variables of the Basic_Item class and derived subclasses. The first group of methods enables access to the instance variables of the objects. The second group corresponds to the basic editing operations that can be performed on the graphi-

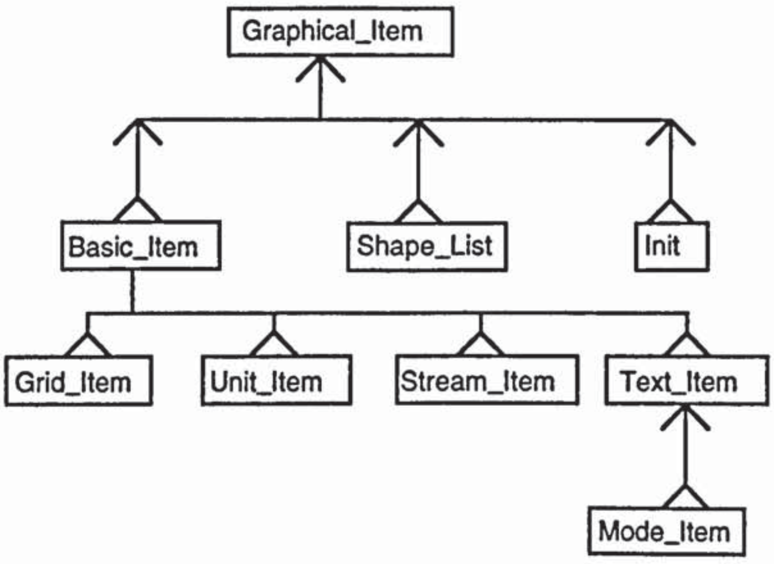


Figure 6.11: Standard graphical item class hierarchy.

cal items provided by the prototype. Some operations, like Move and Resize, use algorithms which will not be described here; see source code for details. Methods such as In_Selection_Area and In_Resize_Area are used to determine if mouse input has been received in certain areas of the graphical item, which result in selection of the graphical item and possibly trigger operations like Move and Resize. Method Check_Connections checks the links between the graphical items, and method Translate translates each graphical item into its corresponding models, views and controllers. Finally, Load_Data and Save_Data enable the user to load and save *shapes*, i.e., pre-defined collections of graphical items, from files.

An instance of the class Shape_List is, basically, a linked list of graphical items, which are currently being used to define the configuration of the physical system on-line. The Shape_List class therefore includes methods to manage and handle the operations associated with the list. Fig.6.13 represents the protocol and instance variables of the Shape_List graphical class. Fig.6.14 represents the protocol of the Init graphical class. The Init class contains only the constructor and destructor, which deallocates memory assigned during the creation of an instance of that class. The constructor initializes the X Window connection to the server and creates all the X and XView resources necessary to the rest of the application such as colormaps, cursors, fonts, etc.. The Init class is therefore

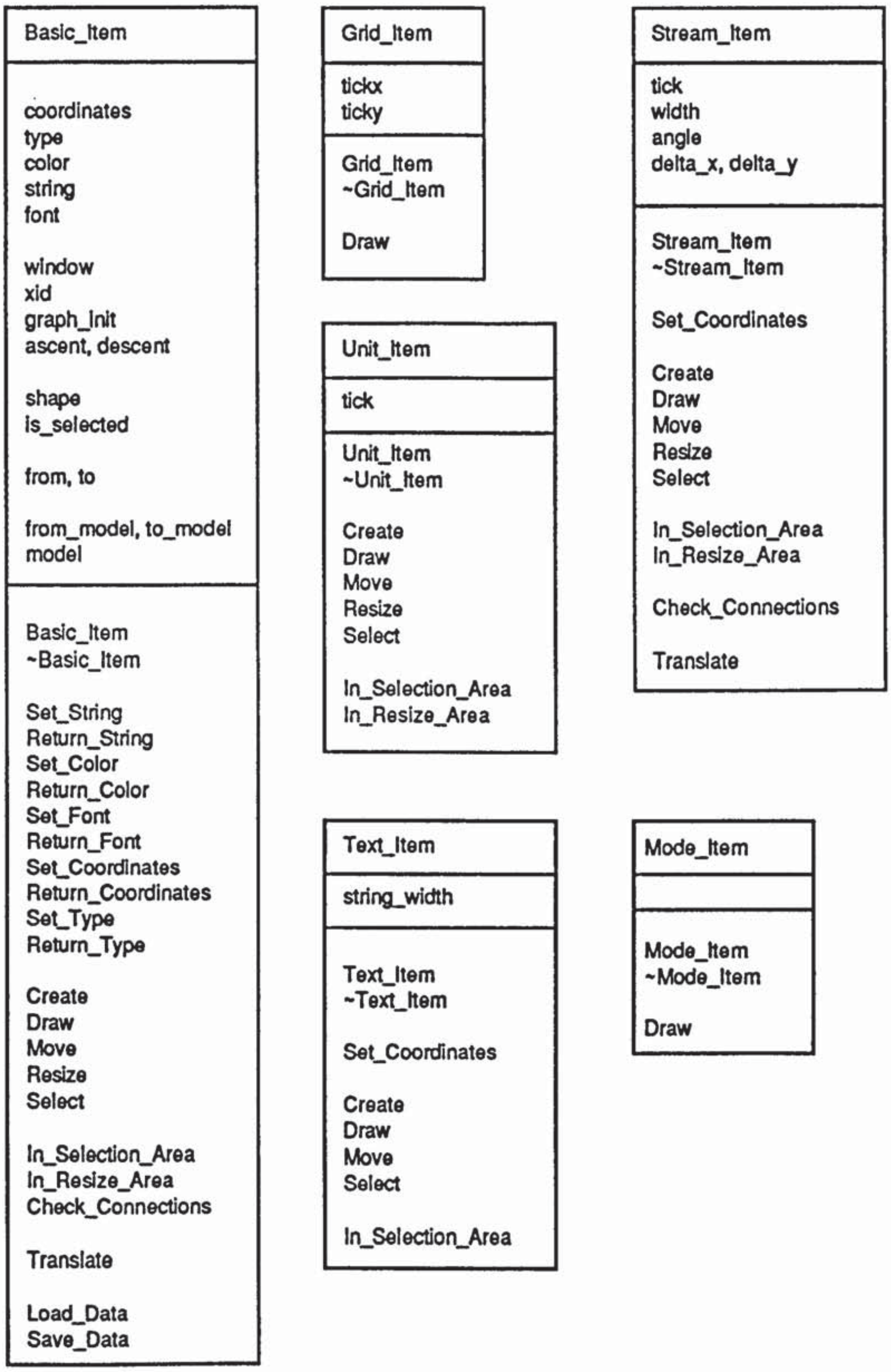


Figure 6.12: Protocol and instance variables of the Basic_Item and derived classes.
(~ denotes the *destructor* method).

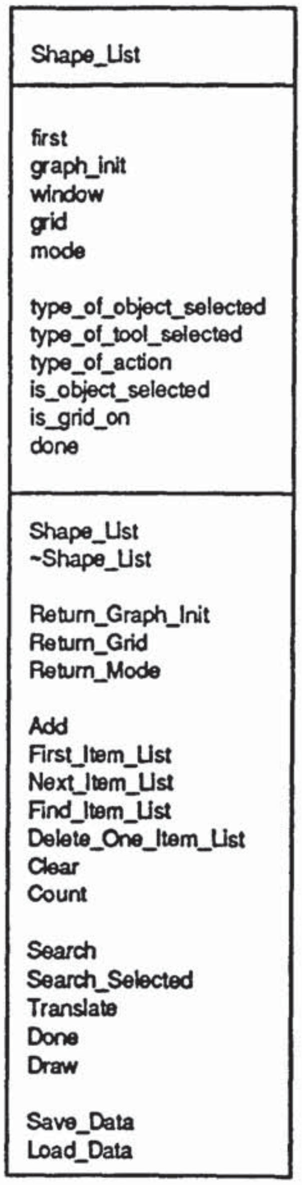


Figure 6.13: Protocol and instance variables of the Shape_List class. (~ denotes the *destructor* method).

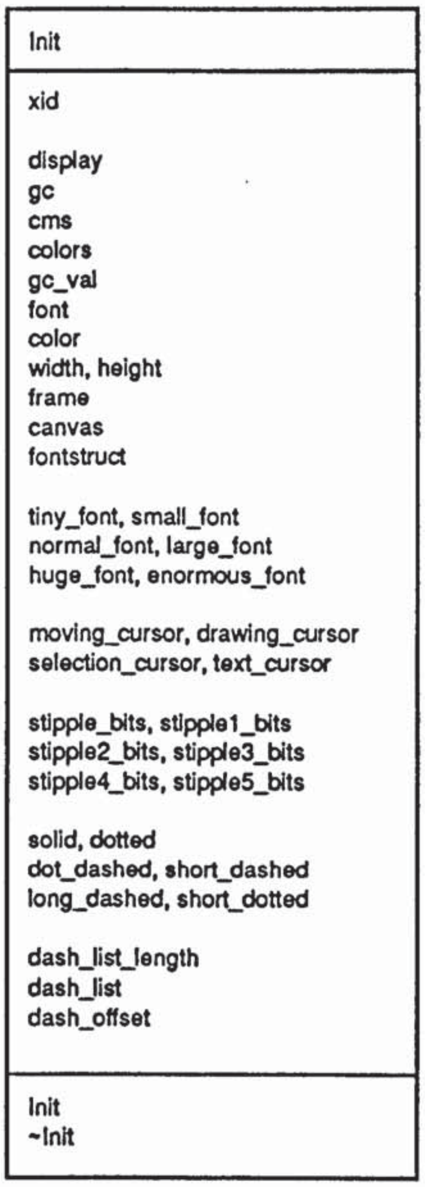


Figure 6.14: Protocol and instance variables of the Init class. (~ denotes the *destructor* method).

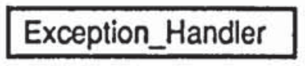


Figure 6.15: Standard exception handler class hierarchy.

a class which initializes and supports the use of graphics and is used intensively by the graphical items and view classes.

6.2.6 Exception Handlers

Only the top, abstract base class was included in the standard library for the exception handlers (see Fig.6.15). Fig.6.16 represents the protocol of the standard exception_handler class. The base class for the exception_handler hierarchy thus contains empty methods to handle all Unix signals. Subclasses can then be created where the methods that catch the signals of interest are overridden.

6.3 Control of the Execution Status of the Simulation Processes

In the Unix operating system, if a user process asks to be informed about asynchronous I/O events, it is informed of such events by Unix signals. Upon reception of a Unix signal, execution jumps to a procedure normally called the *signal handler* for that event. When the signal handler has completed, execution of the process resumes where it left off. Because simulation calculations are especially prone to the occurrence of exceptions resulting from numerical problems (e.g., combinations of parameters which lead to unstable integration and eventually to floating point exception), an exception handling strategy was implemented in the prototype in order to recover from and alert the user of such occurrences. The strategy followed in this work consists of installing exception handlers in all the processes, both master (interface) and slaves (simulation processes). Whereas these functions are performed by an object on the interface side, the language used for the implementation of the slave(s) determines how these functions are performed.

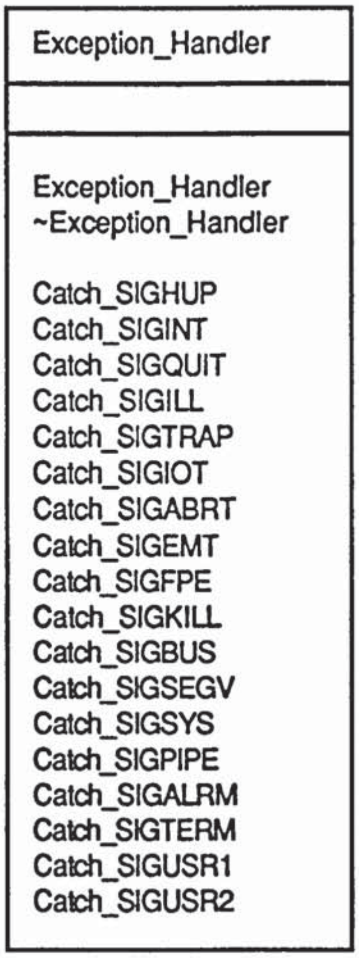


Figure 6.16: Protocol and instance variables of the standard exception_handler class.
(~ denotes the *destructor* method).

The remote processes are therefore able to catch Unix signals generated during execution, such as normal or abnormal termination, the occurrence of floating point exceptions, etc.. Exceptions are caught by the exception handlers of the remote processes and a UNIX signal is sent to the interface using a PVM function, where it is caught by the exception handling object and the appropriate method is executed. The exception handler of the remote process also sends the identity of the process to the interface, as a message, so that the interface knows which remote process caused the exception to occur.

6.4 Conjunction with X and XView

The prototype developed in this work consists of a layer of software which has been superimposed on the X and XView software. It uses X and XView, albeit encapsulated in an OOP environment, intensively and must therefore provide compatibility with the functions they provide and the mechanisms they are based upon. Because X and XView are written in C, some problems occurred and had to be overcome.

6.4.1 Subclassing Methodology

In XView terminology a *canvas* (drawing area) is an object which belongs to a specific package, or class. As has been mentioned already, canvases take as attributes (among other things) pointers to functions which are registered as callbacks and executed if the corresponding event is generated. If an instance of the Canvas class is resized, the appropriate event handler is executed (CANVAS_RESIZE_PROC). Each instance will therefore respond in a different way, depending on the callback registered with the specific object. This implies that instances of the same class have, in practice, different methods. It also holds for objects from other XView packages.

Two approaches are now possible for the definition of object behaviour. One possibility would be to use the standard classes directly, create instances of those classes, and send a pointer to the desired function as a parameter during the creation (or possibly later) of the view. This mechanism is advocated by Menga *et al.* (1991) and is described as the run-time definition of methods. It customizes the behaviour of objects having the same data structure, avoiding the need to define a new class only to reimplement one method. It is employed by X to pass function attributes and leads to a high number of procedures which are not bound to any object or class. Although possible in C++, this situation should be avoided. Therefore, the standard view classes contain empty methods for, for example, the generation of the corresponding panels. The user must create new subclasses of the standard classes where these methods are overridden by the desired procedures. Methods are not passed as function attributes, neither are they re-defined at run-time. For example, two different diagram views (a diagram view presents a schematic representation of the physical entity being simulated) belong to different classes, since although both have the same type of methods (draw the diagram), the methods are in fact different (they draw different things). Although this leads to the generation of a higher number of subclasses, some of which may have only one instance, both the structure of the program and behaviour definition of the objects become clearer.

6.4.2 Pass-through Procedures

The views receive, as the visible elements, user input, which must then be re-directed to the view's controller. In X programs, when user input is received by the application, the callback procedure registered with the object for that event is called, as mentioned above. Because X toolkits are not written in an object-oriented language, some problems occur. One of them is that callback routines cannot be *methods* of objects. This means that it is not possible to register directly, as the procedure to be called when an event occurs on a certain view, any handling procedures that belong to its respective controller. This led to the need for a series of pass-through event-handling procedures, which are used as callbacks, and are automatically registered when the corresponding view is created. The pass-through (which is therefore a free, non-bound procedure) retrieves the controller associated with the view and calls the appropriate method. In order for this to be possible, the controller must be attached to one of the parameters which is sent to the event handler; extensive use of the XView XV_KEY_DATA attribute, which enables data to be attached to any XView object, was therefore made. For example, when a canvas is created, the controller of the view it belongs to is attached to the canvas, so that it can be retrieved later. This implies that special care must be taken concerning the order in which objects and widgets are created, so that the data attached actually refer to existing objects.

Pass-through procedures are transparent to the user, i.e., they have no function other than to redirect user input to the appropriate method of the view's controller. They are the only non-bound procedures of the software developed. Installation of the callback routines in the standard objects is made by the standard views. Installation of these in customized views are the responsibility of the user.

The implementation of exception handling objects presented the difficulties referred to above, since the event handling routines cannot be directly installed as methods of classes. This means that a further pass-through procedure was needed, which is executed when an Unix signal is received by the interface, and in turn calls the appropriate method of the event handling object. When an exception handling object is created, it must install the signal-handling passthrough procedure (a non-bound C procedure) for all the exceptions of interest. It will therefore not change the default event handlers for the other exceptions. Then, depending on the type of exception that the passthrough receives, it will call the appropriate method of the exception handler.

6.4.3 Object Deletion

Because objects are created and deleted at run-time, all the memory allocated for an object must be returned to the heap at the time of its destruction. The C++ *destructors* must therefore ensure that they delete all the data (including other objects) for which memory has been dynamically allocated by the object being destroyed. Because the prototype is used in conjunction with XView, the *parenthood relationship* specified by XView must be respected. The XView objects that compose a view are normally owned by the base frame of that view (unless otherwise specified). Rather than calling xv_destroy (the XView function for deallocating memory) for all the widgets that compose a view, a single call for the base frame will propagate down to the descendents and destroy them all (Heller, 1993). Objects which have other owners (e.g., server_images are owned by the server) must be destroyed separately. Destructors must be carefully implemented since C++ provides no garbage collection mechanism and in this type of application, where objects are constantly created and destroyed at run-time, an untidy management of the allocated memory may lead to out-of-memory situations.

6.5 On-line Definition of the Physical System

Using the classes presented above, a drawing tool has been developed, the "Process Builder" (see Fig.6.17), which enables the user to define the system graphically, choosing the desired units and streams from a displayed menu. Most GUI's employ icons, which represent real-world objects, to represent the elements from which the system model is constructed. Such elements may be elementary, such as a resistor or an integrator, or complex, for example a distillation column or steam turbine (Zobel and Lee, 1992). Each *unit* or *stream* will later translate itself, upon reception of the Translate message, into a *model* object and the associated *view-controller* pairs. The main window of the "Process Builder" is a view of the super_model. In the prototype, defining a physical system proceeds in two stages, namely composing the graphical representation of the system (which corresponds to the creation and manipulation of graphical items) using standard drawing operations, and the translation of the resulting shape into models, views and controllers.

The basic operations provided by the "Process Builder" are similar to the ones necessary to create diagrams graphically (Dolado, 1991): addition of a symbol (in this case a *unit*), and addition of a relationship (a *stream*) between symbols. The reciprocal operations are the deletion of a relationship, which implies the deletion of a coupling (or maybe graphical rearrangement) and the deletion of a symbol.

The icons on the left represent all the major editing operations such as SELECT (thick slanted arrow), ERASE (eraser), access to the available units (rectangle icon), access to streams (thin arrow) and access to text (A icon). "At-

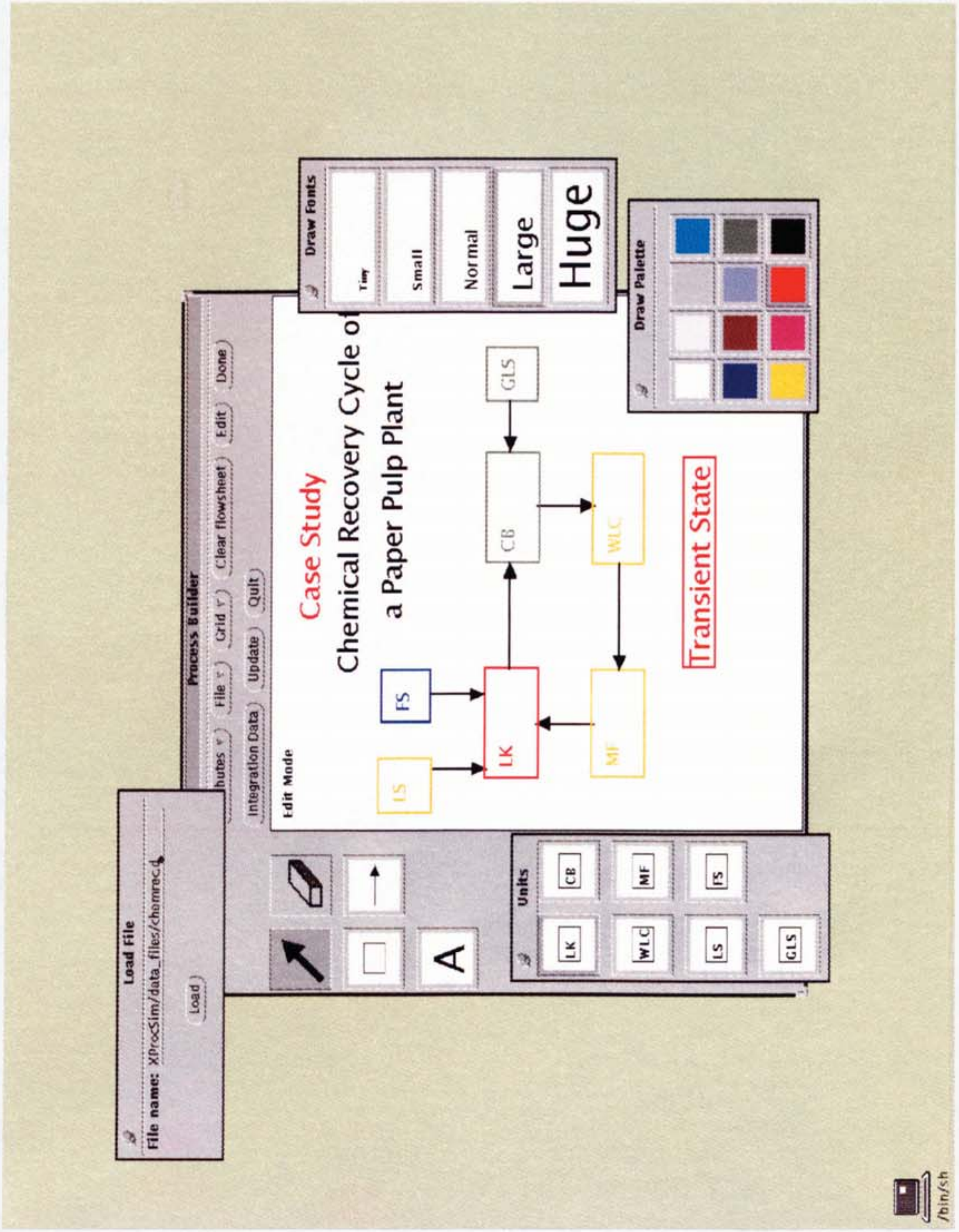

File name: XProcSim/data_files/chemrec.d

Figure 6.17: The Process Builder.

tributes" gives access to the color and fonts views, which enable the user to change the color or text font of graphical items currently displayed. "File" gives access to the Load and Save_File_Views, which enable the user to load a shape (collection of graphical items) or save the current shape. "Grid" enables the user to show or hide the grid and "Clear Flowsheet" deletes the current shape. Two interaction modes (explicitly indicated in the drawing area) exist: Edit Mode and Simulation Mode. "Edit" is the default mode. When the shape represents the desired system, "Done" translates the current shape into its corresponding models, views and controllers, and the system is ready for simulation. Editing operations are disabled at this stage and Simulation Mode is entered. To go back to Edit Mode, which will be necessary only if the user wishes to change the configuration of the physical system, the "Edit" button must be pressed. The "Integration Data" button displays a view, belonging to the super_model, that contains the data corresponding to the integration process (time step, etc.). "Update" updates the state of the system by one time step. This is done by the super_model which sends updating messages, in turn, to all the units, waits for reply messages when completed, and then repeats the process with the streams. When the "Done" button is pushed, the message Translate is sent to all the graphical items in the current shape. Each graphical unit or stream item then creates its corresponding unit or stream model, view(s) and controller(s). The interactions between the several models are determined by their position in the drawing area, so that different configurations of the system can be created. From this point on, in Simulation Mode, graphical display of results is handled by the views just created. Finally, "Quit" deletes all the current models, views and controllers; sends termination signals to the processes created (if any) and finishes the connection to the server. All the model objects possess an *access menu*, associated with the respective primary view. The access menu is a pop-up which displays itself by clicking the Menu mouse button on the graphical representation of the object and gives access to all the views of that object. It must be created by the user, according to the types of views desired for a specific unit or stream. These views may include schematic representations of all units, display of instant internal profiles, design characteristics of the units, physico-chemical data, and others (see Chapter 7).

The translation step is important for two reasons. The first one is that there is a considerable economy in time since the translation is performed only once, when the system is defined, rather than after every graphical operation. Some of the translations are time-consuming because the model associated with the graphical item may have several complex view-controller pairs. Furthermore, the translation operation is the ideal time to perform consistency checking, i.e., to determine if the system represented graphically is mathematically and physically meaningful (e.g., there may an unallowed connection between two units).

To summarize, the prototype developed in this work can be described as a fusion of simplified versions of a graphical user interface framework and a simulation language. It is composed of a set of domain-independent model, view and controller classes. In addition to the widgets provided by the toolkit used, other classes include exception handlers, model-view adaptors and graphical classes. In order to join the C++ layer with the underlying X and XView software, several problems had to be overcome. Application of the prototype to a specific domain area consists of the creation of new classes, subclassed from any of the standard classes. A case study, in which the prototype has been applied to the interactive simulation of the chemical recovery cycle of a paper pulp mill, is described in the next chapter.

Chapter 7

Case Study: Dynamic Simulation of the Recovery Cycle of a Paper Pulp Plant

7.1 Introduction

Process simulation is one of the most widely used tools for the design, optimization or simply the study of the behaviour of industrial units in steady or transient states. The attention devoted to the chemical recovery cycle of kraft paper pulp plants has increased in the last decade due both to the need for energy economy and to comply with stricter environmental protection laws. Because of its practical interest, and also because, due to its complexity, it is representative of the problems posed by interactive simulation, the chemical recovery cycle was selected as a case study for the software package developed in this work. The prototype was therefore applied to the on-line construction of an interface for the dynamic simulation of this industrial process. Detailed dynamic mathematical models were developed and implemented for the main units involved in the cycle. In some cases, this corresponded to mathematical problems of large dimension and difficult solution, leading to long execution times. The management of the large amount of data involved was another factor to take into consideration, as well as the automatic preparation of the results for graphical display.

In this chapter, a brief description of the industrial process is made. The application of the prototype to a specific problem corresponds, in practical terms, to the creation of user-defined classes, sub-classed from the base classes listed in Chapter 6, which have been referred to as *standard* classes. The extension of the standard class library in such a way is described, taking into account the objectives to be attained by the resulting graphical user interface.

7.2 Industrial Process

The manufacture of paper pulp consists basically of the transformation of wood into a fibrous mass, which requires the rupture of the bonds that form the structure of wood. Mechanical, thermal or chemical methods can be used. Among the chemical methods, the most widely used are the kraft, the soda and the sulphite processes. The kraft process corresponds to over 60% of the total north-american production (Smmok, 1986). It is composed of the following operations:

Preparation of Wood

The bark is removed, the wood converted into chips and the chips sieved.

Cooking of Wood

The chips are sent to the digestor where cooking (dissolution of hemicelluloses and lignin) with white liquor (aqueous solution of sodium hydroxide and sodium sulphide) takes place. The black liquor which results from this operation is later separated from the pulp. The pulp itself is washed and undergoes bleaching and finishing operations.

Recovery of Chemicals

The loss of sodium is compensated for by addition of sodium sulphate in the recovery boiler, which is the reason why this process is also designated as the *sulphate* process. The black liquor resulting from the cooking of the wood is burnt in the recovery boiler, producing the smelt (molten inorganic compounds, mainly sodium carbonate and sodium sulphide). The smelt is dissolved immediately after leaving the recovery boiler forming green liquor. At this stage, the green liquor contains a considerable amount of suspended solids (dregs) consisting of insoluble inorganic compounds (compounds of heavy metals, silicates and aluminates from the refractory lining of the boiler and, on a smaller scale, fine carbon particles) and needs to be clarified. After clarification, the green liquor is made to react with quicklime (calcium oxide) from the lime kiln, in order to regenerate the white liquor. To carry out this operation, a slaker is used where the calcium oxide is hydrated (slaking reaction) and where the transformation of the resulting calcium hydroxide into calcium carbonate (causticizing reaction) begins. The reacting slurry goes through a classifier where the larger particles (grit) are removed and then through three or four causticizing tanks where the causticizing reaction approaches equilibrium. There is an upper limit to the efficiency of the conversion due to the reversibility of the causticizing reaction. The white liquor is then separated from the calcium carbonate in a clarifier in order to achieve an adequate degree of clarification and the lime mud — basically a suspension of calcium carbonate in water — is washed and thickened. The thickened lime mud is calcinated in a lime kiln where the calcium carbonate is converted into calcium oxide so that it can be used in the slaker again. Fig.7.1 illustrates the sequence of operations involved in paper pulp manufacture. The continuous nature of the process involves a high number of unit operations: filtration, classification, mixing, sedimentation and storage and transportation of solids, as well as storage and pumping of liquids. It is sometimes useful to visualize the chemical recovery process as a closed loop operation, where solids are involved in one cycle and the liquor in another (see Fig.7.2). Contact is promoted between the solids and the liquor in order to produce the white liquor used in cooking. Each of them is later recycled and reconverted into the required chemical forms. The loop has become, over the past few years, progressively more closed, due to economic and environmental requirements. This fact, which aims to reduce the amount of polluting effluents, results in a build-up of undesirable compounds in both cycles (Tran *et al.*, 1990). A continuous addition of make-up lime, usually in the form of limestone, is necessary in order to compensate for constant losses in the slaking/classification and calcination operations, and constitutes one of the sources of impurities. Figs.7.3 and 7.4 show the general arrangement of the equipment of the chemical recovery section and a side view of the causticizing tanks, respectively. The approach used in this study was to develop detailed mathematical models for the main units involved in the cycle. Only a simplified version of the cycle itself was considered (see Fig.7.5).

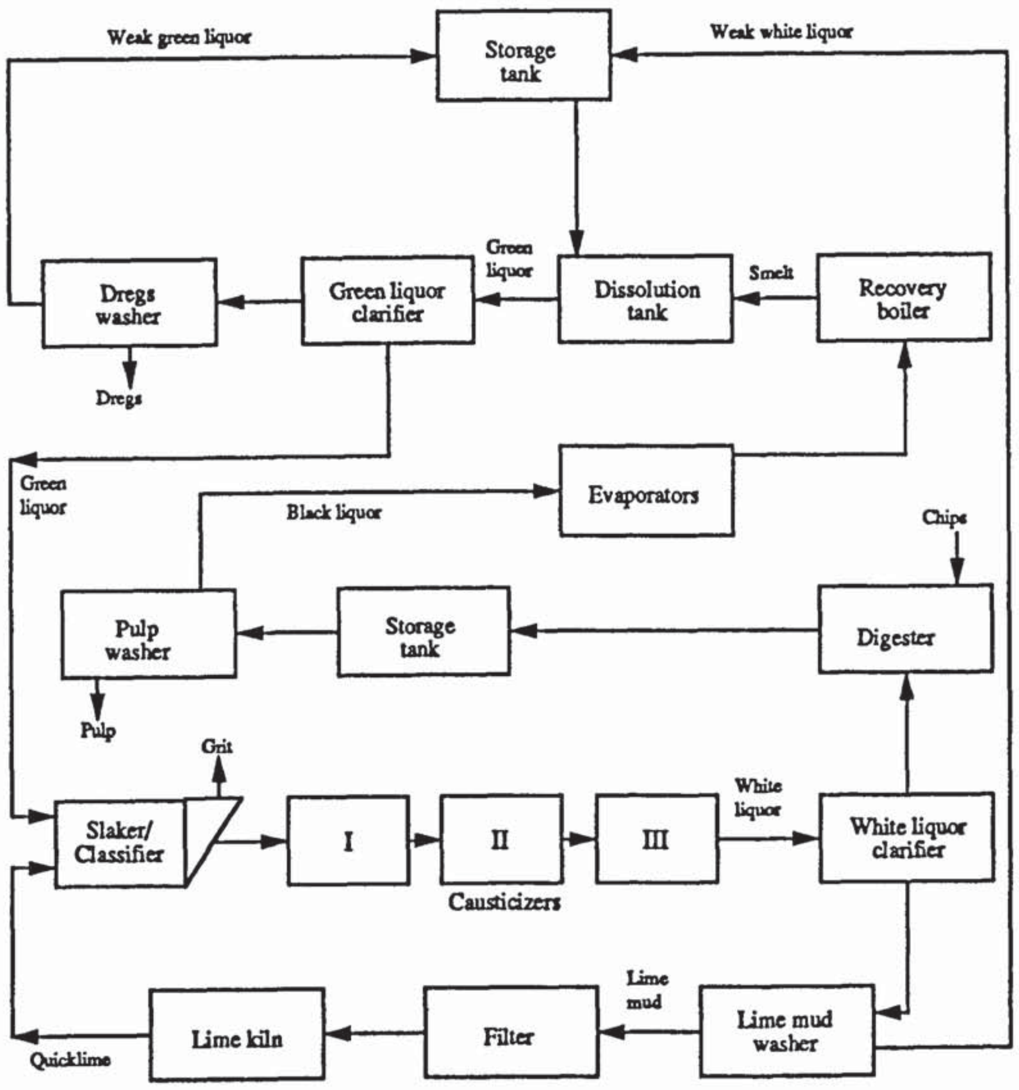


Figure 7.1: Simplified diagram of the kraft process for the manufacture of paper pulp.

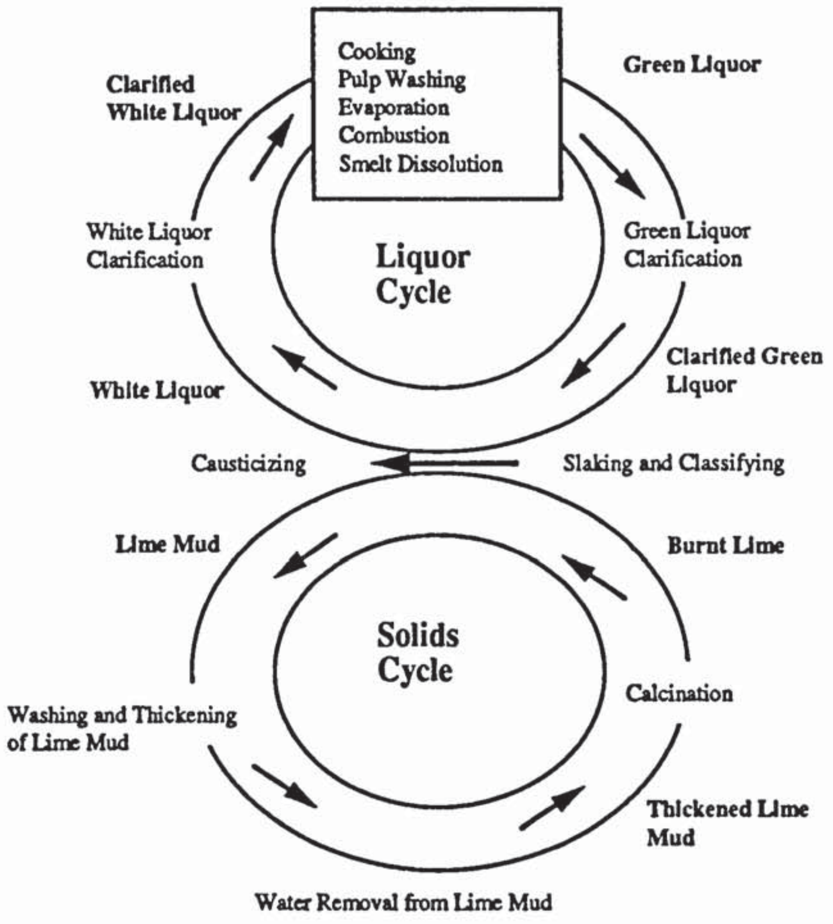


Figure 7.2: Cycles involved in the chemical recovery process.

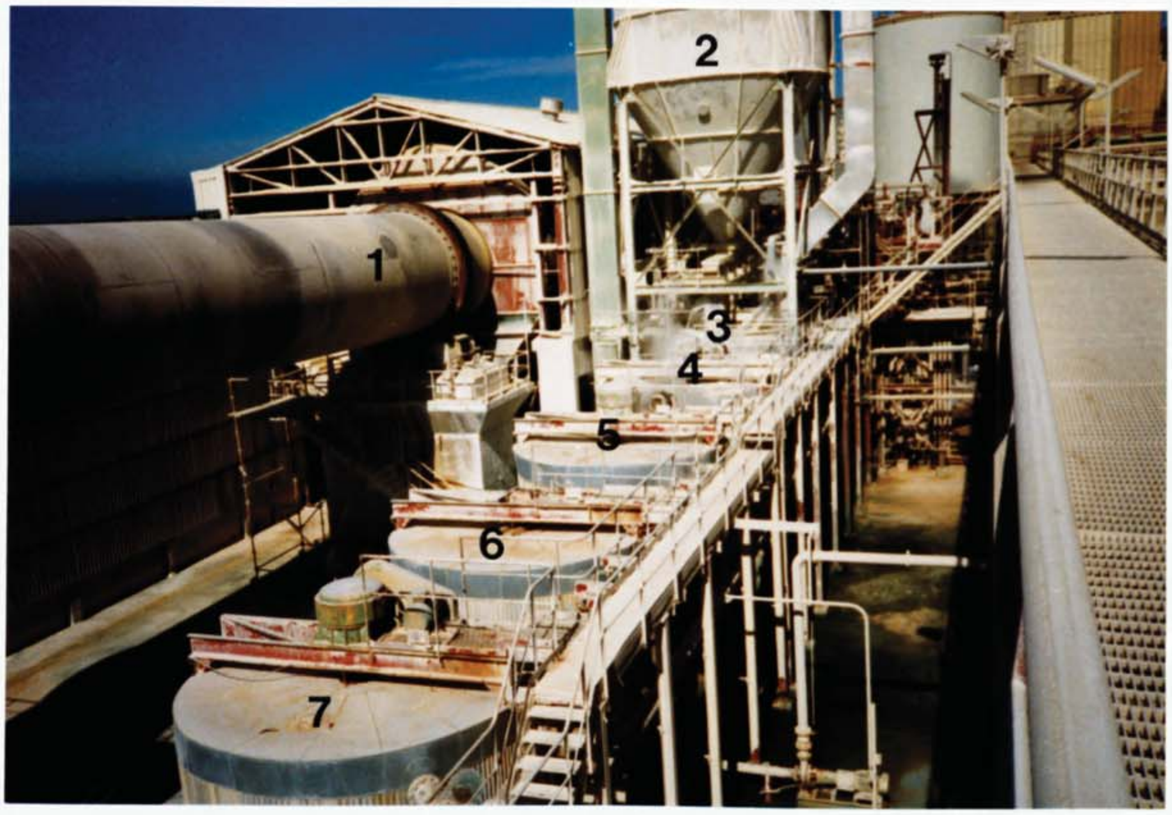


Figure 7.3: General arrangement of the equipment in the chemical recovery zone. (Courtesy of SOPORCEL — Leirosa, Portugal)

1 - Lime Kiln; 2 - Lime Bin; 3 - Slaker; 4, 5, 6 and 7 - Causticizers

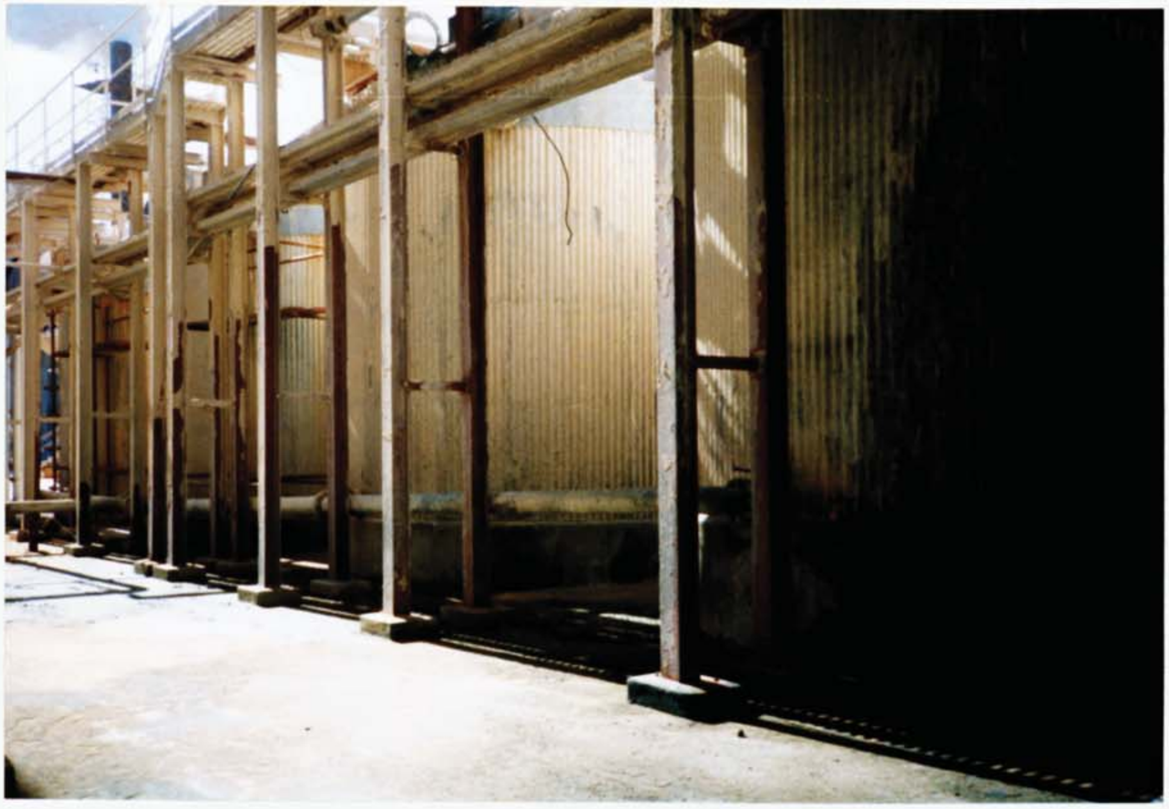


Figure 7.4: Side view of the causticizing tanks.
(Courtesy of SOPORCEL — Leirosa, Portugal)

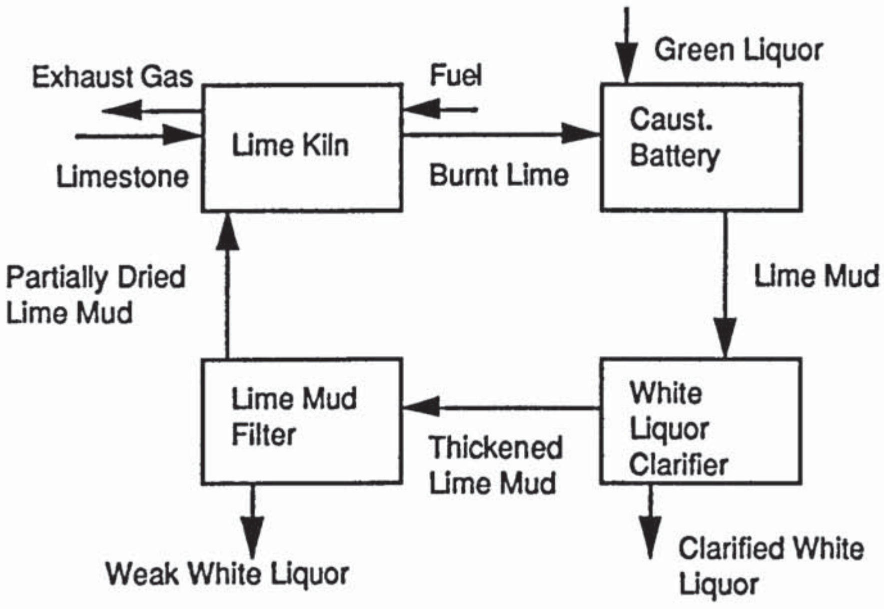


Figure 7.5: Simplified diagram of the chemical recovery cycle of a paper pulp mill.

A mathematical model for the steady state of the lime kiln was implemented, based on a previous model found in the literature. A dynamic model for the transient state of the lime kiln, the most important unit energetically due to fuel consumption, was developed and implemented (see Appendix A). A microscopic model for the solid particles undergoing the causticizing reaction was also developed (see Appendix B, section B.2.1), and used in a detailed model for the steady-state of the causticizing reactors (see Appendix B, section B.2.2). Because the simulation in such detail, in transient state, of the causticizing battery does not seem to be feasible with current computational means, an approximate model was developed and used in the simulation package (see Appendix B, section B.3). A dynamic model was also developed and implemented for the white liquor clarifier (see Appendix C). Finally, a simplified model was implemented for the lime mud filter, in which it was considered to be in steady-state at all times (see Appendix D). Previous attempts to simulate the chemical recovery cycle as a whole used either very simplified models for all the units (Jacobi and Williams, 1973a; Jacobi and Williams, 1973b) or a totally different approach in which transfer functions were used to simulate the response of the system and the gains and parameters were determined experimentally (Uronen *et al.*, 1976;

Uronen and Aurasmaa, 1979). The collection of models presented in Appendices A through D represents the most detailed approach yet to the simulation of the chemical recovery cycle as a whole. All the mathematical models were implemented in Fortran77 and integration of the differential equations in time was done using the LSODI routine (Hindmarsh, 1980), which uses the BDF (Backward-Difference Formula) for integration of stiff differential equations.

7.3 Specification of the Graphical User Interface

The enforcement of a structure for the application, the on-line definition of the physical system and the control of the simulation are domain-independent and are provided by the standard classes of the prototype. In the following step, which is the application of the prototype to a specific system, it is first of all necessary to determine the objectives that the resulting interface must attain. This will, for example, determine the type of new views to be developed and the functions they must provide. The objectives aimed at are therefore domain-specific. In this case, they include the graphical representation of the process and individual units, the organization and easy access to simulation and numerical data, the graphical representation of results, and the possibility of using some form of concurrency, whilst retaining control of the status of all the simultaneous processes. Other desirable features, such as the implementation of an on-line help system and error warning and recovery (simplified versions of which were included in the first version of the prototype), are not the direct objective of this research. In this section, a brief description of the requirements specification is made, together with the solutions found.

Graphical Representation of the Process Model and Individual Models

An essential requisite is the existence of a graphical representation of the model of the physical process, where graphical items symbolize the model objects into which the overall model has been decomposed. As mentioned before, the primary view of the super_model is a non-interactive, non-visible view, which at present is the owner of all the new views created at run-time, and which may in the future be attributed other functions as well. The secondary view of the super_model, the draw_view, is an instance of class Draw_View, and consists of an interactive diagram (in this example, a very simplified flowsheet) of the process, which therefore displays the current configuration of the process model.

More elaborate symbols can be used to represent the units, although the implementation of the basic editing operations like Move and Resize, currently supported, will in this case be considerably more complex; graphics primitives such as the ones provided by GKS (Peddie, 1992) would be extremely helpful for this purpose. X Window, *per se*, offers very little support to structured graphics (see list of graphics primitives in Nye (1990, 1990a)). The graphical classes may also be extended to include grouping and composition of graphical objects, possibility of moving them to the "back" and "front" of the diagram, etc., all of which are normally found in specialized drawing applications such as MacDraw for the Macintosh.

No limitation must be imposed on the number, or type of views, for each model; among these are the schematic representation of the industrial units, display of design dimensions, display of physico-chemical properties used in the simulation, graphical representation of current variable profiles (in the case of *distributed systems*, modelled by partial differential equations), or evolution of selected variables in time. A number of view classes must therefore be developed that performs these functions; see Fig.7.9 for a list of the new view classes.

Furthermore, since these views are created at run time, a means to access them, also dynamically created, must be provided. The draw_view, which contains the information concerning the overall model, can be used to achieve this objective. After the translation step, each graphical item that composes the diagram displayed in the draw_view knows the model it corresponds to. Pressing the Menu button of the mouse over the graphical representation of a unit (or stream) displays a pop-up menu (the *access_menu*) which lists and gives access to all the views available for that unit (or stream) model.

Organization and Access to Simulation Data

The data used in the simulation may come from any source, either as default values from a commercial database of physical properties or from data files (which is the current case). However, as mentioned before, all the values used in the simulation must be available and the user must be allowed to change them through the interface. A means to achieve this is to create views that display interactive data *panels*, each of which contains any number of *panel_items* that indicate the name of the data item and the units in which its value is expressed (e.g. (m) or (Kg/s)). Among the data are the above-referenced dimensions of the unit and physico-chemical parameters such as heat transfer coefficients, activation energies and Arrhenius constants for the chemical reactions (see section 7.4.2). This also allows the grouping of the data into different categories, each of which corresponds to a different view, and therefore makes it easier to access the desired value.

Access to Integration Data

The data used in the management of the simulation, such as the time step after which the interactions between the models are reset or the frequency with which the results are written to files, must also be available. Since the super_model is the entity that coordinates the simulation, it is logical that these data be displayed in a view associated to the super_model. Therefore, one of its secondary views must be a data panel that displays and allows the user to change these parameters.

Graphical Representation of Results

The graphical display of the state of the models is an essential feature. The most common types of representation are line and bar charts, so views that provide this type of representation must be provided. Depending on the possible models they may be associated with, and also because even if they are associated with the same model they will display different sets of variables (i.e., different *aspects* of the model) different model-view adaptor classes must also be created to maximize reusability.

Control of the Execution Status of the Simulation Processes

Exceptions occurring during the execution of the simulation processes, especially arithmetic exceptions, must be detected and the user informed. The exception handling strategy described in Chapter 6, sections 6.2.6 and 6.3, enables the remote processes to catch any Unix signals generated during execution, such as normal or abnormal termination or the occurrence of floating point exceptions. This information is then sent to the interface. This enables detection and recovery of errors indirectly — and sometimes unpredictably — caused by the user, such as sending numerically unsound sets of parameters to the simulation processes.

Error Warning and Recovery

If errors are made by the user on the interface side, such as trying to load a non-existent file, error detection is performed by checking the value returned by the respective function. When an error value is returned, a dialog box is displayed (see Fig.7.6). The current implementation of the prototype does not cover all the possible user errors; rather, it exemplifies an effective strategy, similar to the one used by other well-known graphical user interfaces (e.g., OpenWindows and the Macintosh interface).

Additional Features Supported in the First Version of the Prototype

Including some UNIX commands as menu options was thought to be a convenient feature in the first version of the prototype. The aim was to provide a very simple interface to the operating system in order to perform operations such as listing result files or checking the processes currently being executed. If the user wanted direct access to the operating system, a terminal emulator window was created by the interface. However, because the interface will, in any case, need the OpenWindows environment to run, which in itself enables the creation of command tools, shell tools, etc., this feature was considered unnecessary and dropped in the current version.

Also, the first version offered a help system, in which help buttons existed in every window which, when pressed, displayed a text read-only window which

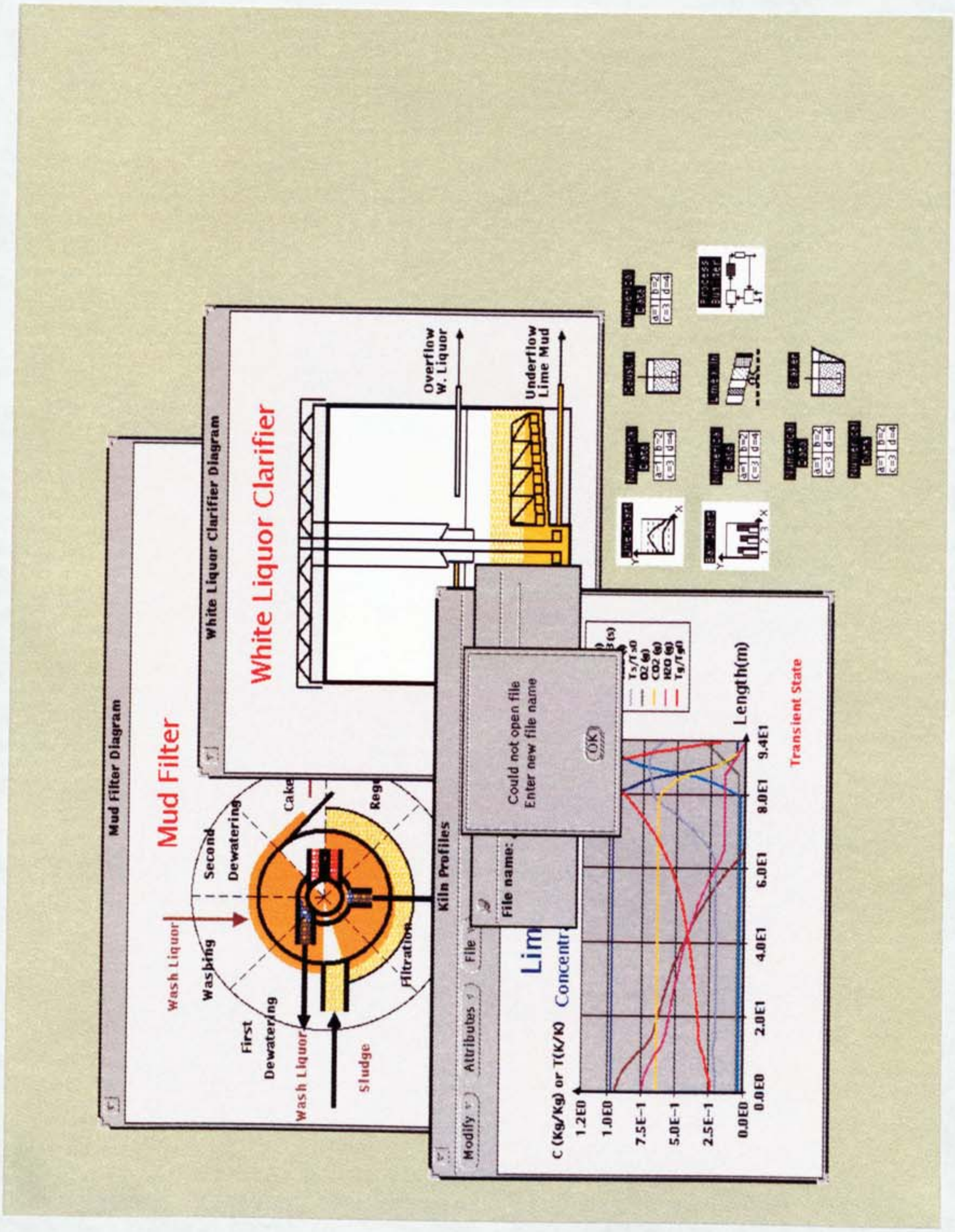

Could not open file
Enter new file name

Figure 7.6: Attempt to load a non-existent data file.

contained summary information. This was a form of context-sensitive help (i.e., help information that is related to the particular point the user has reached), which gives users the option to make help requests only when desired and seems to be better in terms of speed of task and error frequency (Rubin, 1988). However, in order to be effective, the optimal structure of such a help system has to be seriously thought through (and tested) and was considered out of the scope of this work.

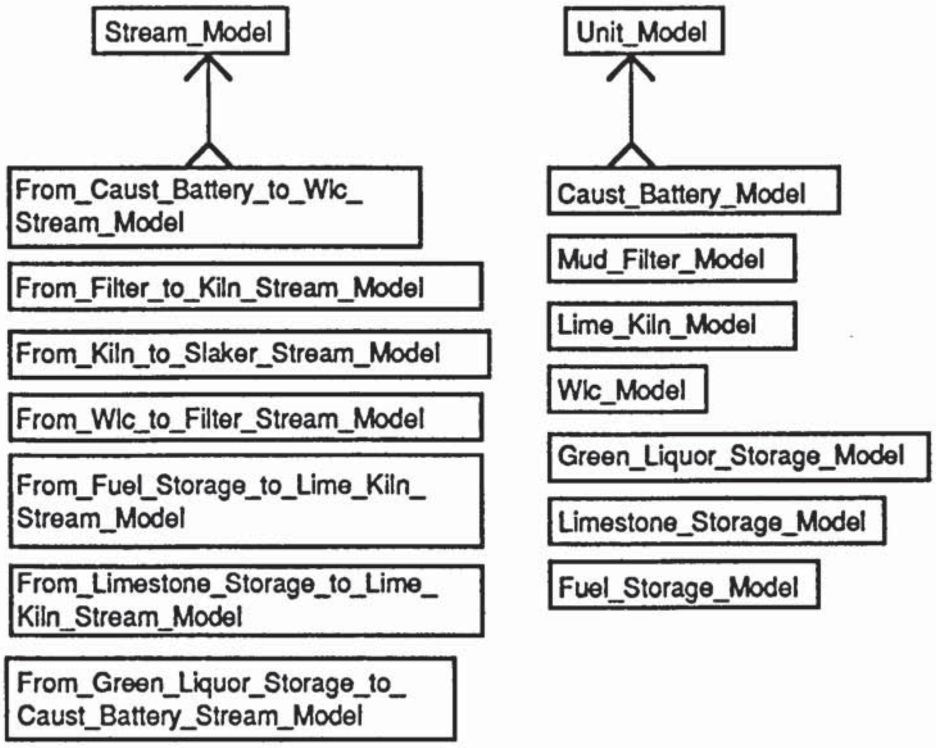
7.4 Extension of the Standard Class Library

The simplified version of this cycle includes the lime kiln, the causticizing battery, the white liquor clarifier and the lime mud filter (see Fig.7.5). Each of these correspond to a different *model* class. All the streams correspond to different classes as well, since they connect units of different classes. Every stream must have a *source* and a *sink*, i.e., it must start in a unit and end in another. In cases where several feeds exist to the same unit, this enables the distinction between the different streams (e.g, in this case we must have a *fuel storage* unit and a *limestone bin* unit).

All the windows are scalable. No limitation was imposed on the maximum number of windows that can be displayed simultaneously. Windows can at any time be reduced to their closed, or *iconic* state (icons have been attributed to all of them), and they can also be dismissed and later retrieved, so avoidance of a cluttered screen was left to the user.

7.4.1 New Model Classes

A number of new model classes has been created using the existing standard classes (see Fig.7.7) They include classes for the simulation of all the units involved in the cycle, as well as stream classes for all the physically meaningful connections between the units. Although the updating methods for the streams are usually quite simple numerically, they can include very complex numerical procedures for the units. The lime kiln, for example, involves the solution of a system of nearly one thousand stiff ordinary differential equations, resulting from the discretization (using orthogonal collocation) of the spatial indepen-



Stream_Model
From_Caust_Battery_to_Wlc_Stream_Model
From_Filter_to_Kiln_Stream_Model
From_Kiln_to_Slaker_Stream_Model
From_Wlc_to_Filter_Stream_Model
From_Fuel_Storage_to_Lime_Kiln_Stream_Model
From_Limestone_Storage_to_Lime_Kiln_Stream_Model
From_Green_Liquor_Storage_to_Caust_Battery_Stream_Model
Unit_Model
Caust_Battery_Model
Mud_Filter_Model
Lime_Kiln_Model
Wlc_Model
Green_Liquor_Storage_Model
Limestone_Storage_Model
Fuel_Storage_Model

Figure 7.7: Problem-specific model classes.

dent variable of hyperbolic partial differential equations. The updating method for the white liquor clarifier involves the solution of a variable number of ordinary differential equations, which implies re-ordering and re-numbering of the vectors for the dependent variables. Moreover, these classes provide concrete implementations of methods where the creation of remote processes is made and the inter-process communication is managed (see Fig.7.8). Note that the updating step is performed, for a certain unit, either locally (in which case the Update method is re-defined and used) or remotely (in which case the last five methods presented in Fig.7.8 are used). These two sets of methods are therefore never used simultaneously for the same unit. For details, see the source code of the respective class implementation.

7.4.2 New View Classes

All the view classes developed assume a colour machine, while retaining the consistency associated to the same default background colour through all the windows (light grey). Where possible, colour was used to convey information

```
Initialize:
    initialize the instance variables of the model (usually
reading data from a file);
   send Update message to the primary view of the model
(which propagates through all the secondary views).

Update:
   call procedure that updates the state of the model;
   send Update message to the primary view of the model.

Create_Process:
   create remote slave pvm process(es) which waits for the
simulation data.

Start_Update:
   send simulation data to the remote slave process(es)
(asynchronous non-blocking send); slave starts computations
after receiving data.

End_Update:
   receive updated data from remote slave process(es)
(blocking receive);
   send Update message to the primary view of the model.

End_Process:
   send Unix signal SIGTERM to the remote slave
process(es).

Send_Signal:
   send a specified Unix signal to the remote process(es)
(including user-defined signals SIGUSR1 and SIGUSR2).
```

Figure 7.8: Functions performed by the methods of the model classes related to the simulation.

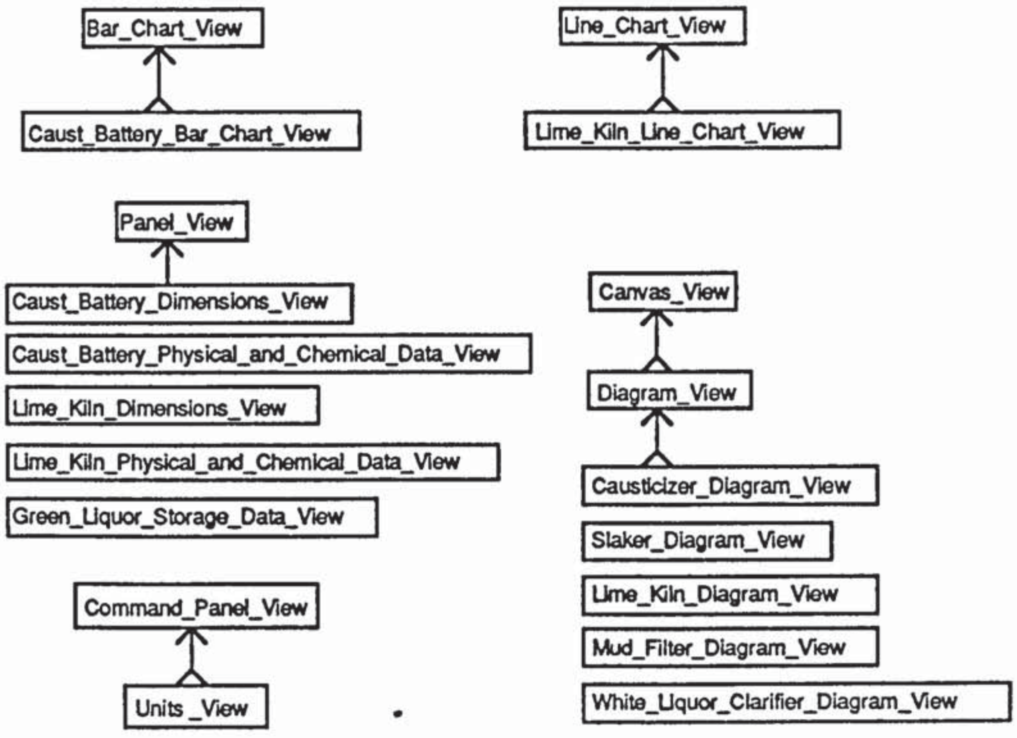

Bar_Chart_View
Caust_Battery_Bar_Chart_View
Line_Chart_View
Lime_Kiln_Line_Chart_View
Panel_View
Caust_Battery_Dimensions_View
Caust_Battery_Physical_and_Chemical_Data_View
Lime_Kiln_Dimensions_View
Lime_Kiln_Physical_and_Chemical_Data_View
Green_Liquor_Storage_Data_View
Command_Panel_View
Units_View
Canvas_View
Diagram_View
Causticizer_Diagram_View
Slaker_Diagram_View
Lime_Kiln_Diagram_View
Mud_Filter_Diagram_View
White_Liquor_Clarifier_Diagram_View

Figure 7.9: Problem-specific view classes.

about the physical system itself. Units where green liquor is processed are coded green, units where lime is the main reactant are coded yellow, and so forth. In the lime kiln diagram, for example, colour goes from yellow to red as temperature increases.

A number of new view classes has been created using the existing standard classes (see Fig.7.9) They correspond to:

a) access to all the parameters used in the simulation (Caust_Battery_Dimensions_View, Caust_Battery_Physical_and_Chemical_Data_View, etc.);

b) "static" views, adequate for novices, presenting schematic representations of the industrial units (all the views subclassed from the Diagram_View class);

c) dynamic views, presenting selected sets of data, which are automatically updated when the calculations of the respective model (integration of differential equations in time) are done (Caust_Battery_Line_Chart_View and Lime_Kiln_Line_-Chart_View;

d) a view which lists, in the form of panel items, all the units currently available in the model class library (Units_View).

The types of panel items used to access the data of the models can be different; for example, sliders may be used instead of data items. It is up to the user to customize the standard views. Figs.7.10 and 7.11 show, respectively, the views corresponding to the lime kiln and the causticizing battery models. When a view is to be used as a primary view, it must both create the access menu and include, in the methods of its controller, the actions to be taken when the menu is used (i.e., display the other dependent views). The classes sub-classed from the Line_Chart_View and Bar_Chart_View simply extend these in a way that these functions are performed.

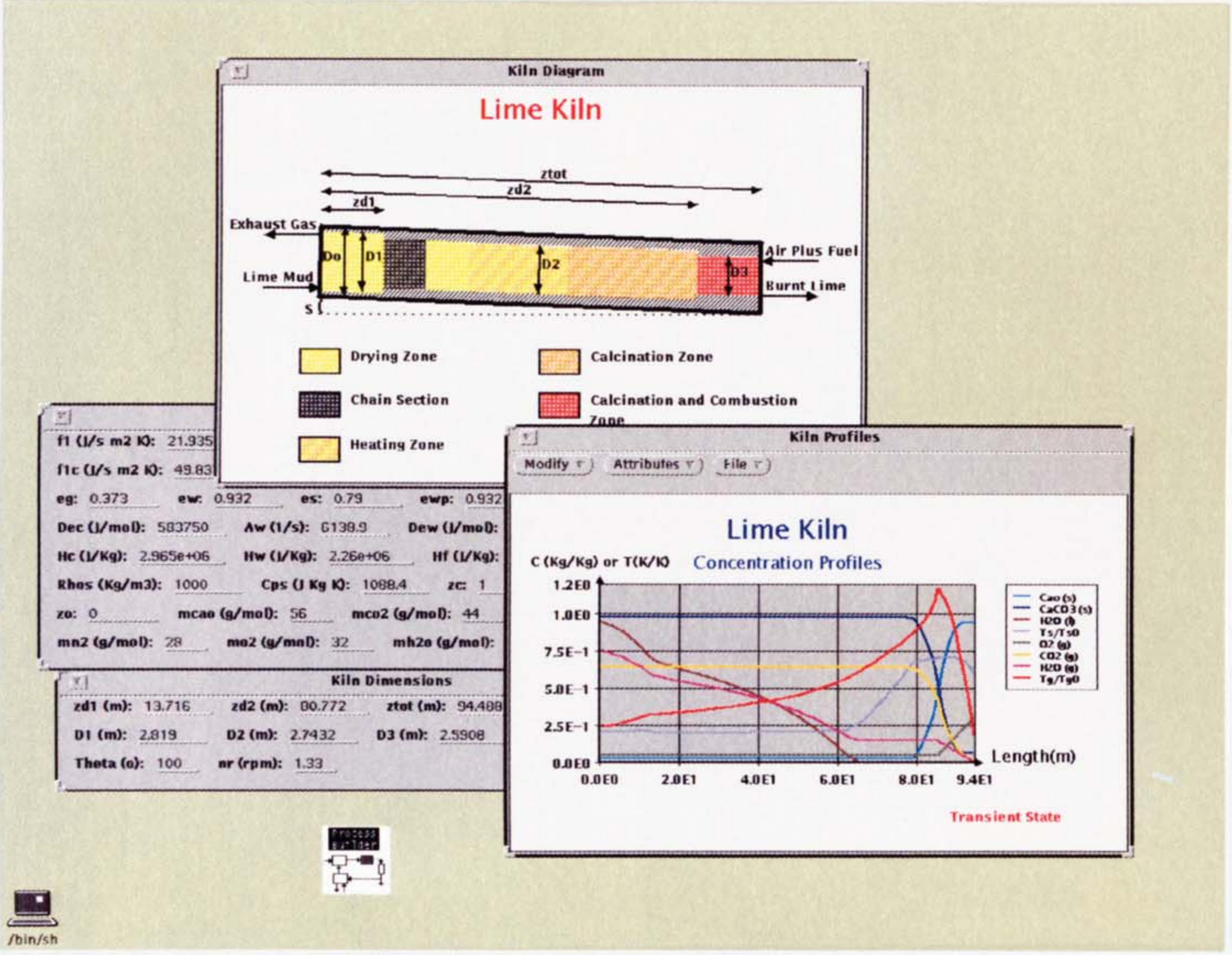
7.4.3 New Controller Classes

A number of new controller classes has been created using the existing standard classes (see Fig.7.12) Note that now more closely related controller classes have been created (e.g., classes Lime_Kiln_Line_Chart_Controller and Caust_Battery_Bar_-Chart_Controller inherit from Line_Chart_Controller and Bar_Chart_Controller, respectively, since they simply add the creation and control of the access menu).

7.4.4 New Adaptor, Graphical Item and Exception Handler Classes

A number of new adaptor, graphical item and exception handler classes have been created using the existing standard classes (see Fig.7.13) The adaptors know exactly which kind of data to retrieve from the model, and which are the view instance variables that depend on that data. They therefore insulate the domain knowledge from the interface. They perform their function (method Convert_Data) only when the Update message is sent to the view.

The new graphical item classes (see Fig.7.13) correspond to the graphical symbols used to represent all the units available for this problem. For simplicity, they all correspond to a rectangle, inside which there is a text code that indicates the type of unit. This means that they share most of the editing operations provided by the standard Unit_Item class. No new stream graphical items are required since their representation is always the same (a plain line with an arrow at the end).

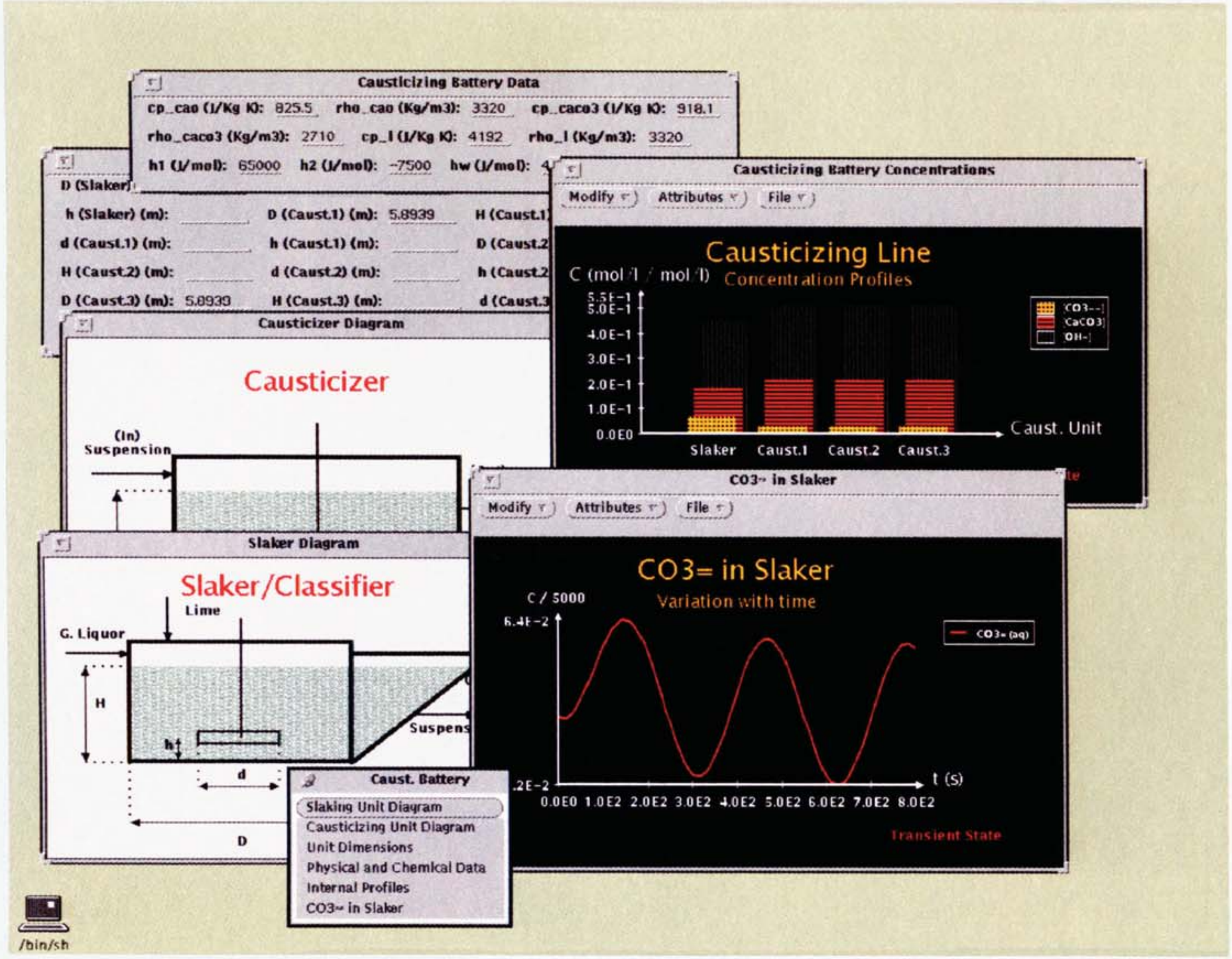


f1 (J/s m2 K): 21.935
f1c (J/s m2 K): 49.83
eg: 0.373 ew: 0.932 es: 0.79 ewp: 0.932
Dec (J/mol): 583750 Aw (1/s): 6138.9 Dew (J/mol):
Hc (J/Kg): 2.965e+06 Hw (J/Kg): 2.26e+06 Hf (J/Kg):
Rhos (Kg/m3): 1000 Cps (J Kg K): 1088.4 zc: 1
zo: 0 mcao (g/mol): 56 mco2 (g/mol): 44
mn2 (g/mol): 28 mo2 (g/mol): 32 mh2o (g/mol):

Kiln Dimensions

zd1 (m): 13.716 zd2 (m): 80.772 ztot (m): 94.488
D1 (m): 2.819 D2 (m): 2.7432 D3 (m): 2.5908
Theta (o): 100 nr (rpm): 1.33

Figure 7.10: Views for the lime kiln model.


Causticizing Battery Data

cp_cao (J/Kg K): 825.5 rho_cao (Kg/m3): 3320 cp_caco3 (J/Kg K): 918.1

rho_caco3 (Kg/m3): 2710 cp_l (J/Kg K): 4192 rho_l (Kg/m3): 3320

h1 (J/mol): 65000 h2 (J/mol): -7500 hw (J/mol): 4

D (Slaker)

h (Slaker) (m): D (Caust.1) (m): 5.8939 H (Caust.1)

d (Caust.1) (m): h (Caust.1) (m): D (Caust.2

H (Caust.2) (m): d (Caust.2) (m): h (Caust.2

D (Caust.3) (m): 5.8939 H (Caust.3) (m): d (Caust.3

Figure 7.11: Views for the causticizing battery model.

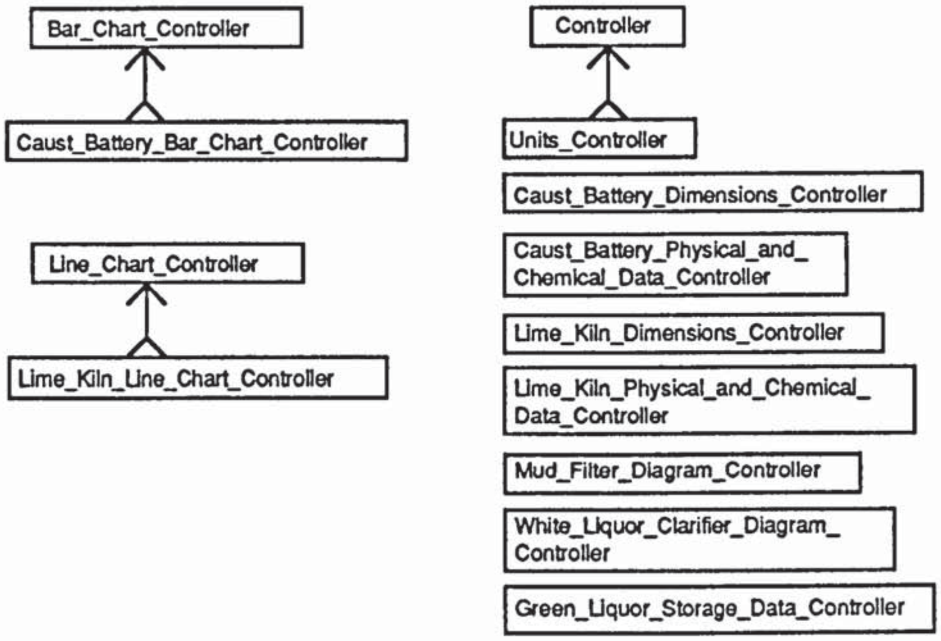

Bar_Chart_Controller
Caust_Battery_Bar_Chart_Controller
Line_Chart_Controller
Lime_Kiln_Line_Chart_Controller
Controller
Units_Controller
Caust_Battery_Dimensions_Controller
Caust_Battery_Physical_and_Chemical_Data_Controller
Lime_Kiln_Dimensions_Controller
Lime_Kiln_Physical_and_Chemical_Data_Controller
Mud_Filter_Diagram_Controller
White_Liquor_Clarifier_Diagram_Controller
Green_Liquor_Storage_Data_Controller

Figure 7.12: Problem-specific controller classes.

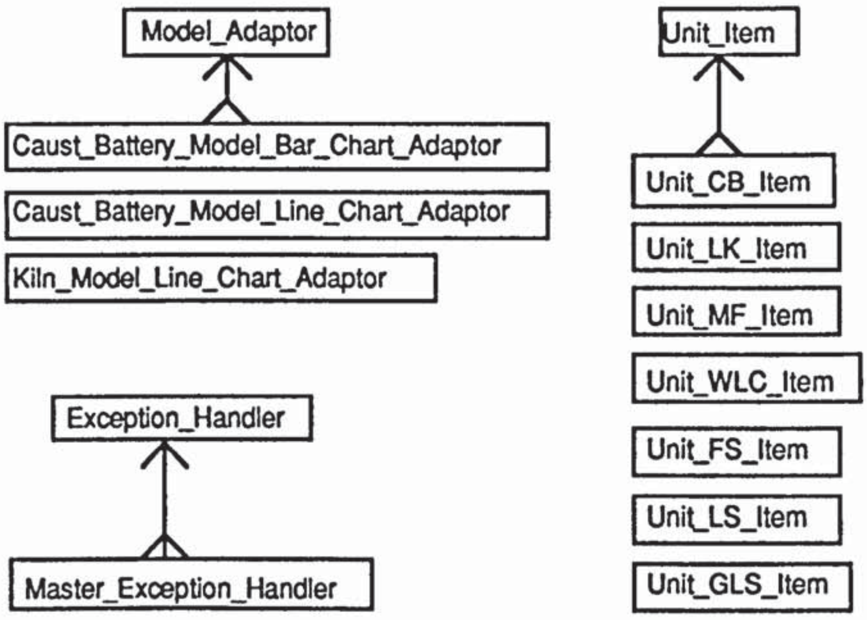


Figure 7.13: Problem-specific adaptor, graphical item and exception handler classes.

Finally, the new exception handler class created (see Fig.7.13) simply defines the exceptions which must be detected in this specific application and what actions must be performed in each case (overriding the methods of the Exception_Handler class).

7.5 Simulation Results

7.5.1 Sample Simulation Results Obtained

Results were obtained for the individual mathematical models used in the simulation and are presented elsewhere (see Pais and Portugal (1993a, b) for results concerning the lime kiln, and Pais and Portugal (1994a, b, c) for results concerning the causticizing battery). Several simulation results for the model used in the white liquor clarifier are presented by Attir *et al.* (1976), although not for the same set of parameters. This type of detailed study will therefore not be repeated here; only a selected set of simulation results are presented, so that both the type of output presented by each model and the complexity of the simulation are outlined.

Fig.7.14 presents the variation of the temperatures in the lime kiln along its axial coordinate, and Fig.7.15 presents the variation of the concentrations along the same coordinate. Since the dynamic state of the kiln is described by a set of partial differential equations (see Appendix A) in space and time, these profiles are time-dependent, i.e., the values represented correspond to a specific moment in time. Colour coding greatly increases the ease of perception; compare Fig.7.15 with Fig.7.10. (The line chart also offers different line styles, and the bar chart offers several fill styles, to enable colour-independent differentiation between the lines and the bars.) Figs. 7.16 to 7.19 present the evolution of the concentrations in the slaker and causticizing tanks during start-up, i.e., when lime starts to be added to the slaker tank, initially filled with green liquor. Notice the delays as the system becomes of higher order (first, second and third causticizing tanks). Fig.7.20 presents a typical variation of the solids concentration with depth in the white liquor clarifier.

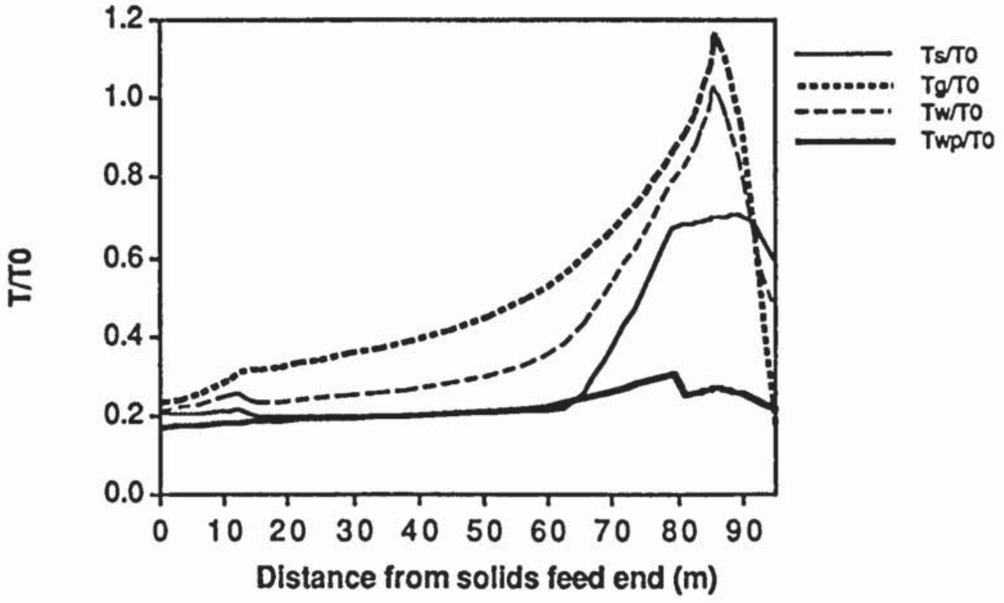


Figure 7.14: Variation of gas, solids, and inner and outer wall temperatures of lime kiln with distance from solids feed end.
(Adimensionalization constant is 1667 K; all temperatures are expressed in K.) Tg = Gas Temperature, Ts = Solids Temperature, Tw = Inner Wall Temperature, Twp = Outer Wall Temperature.

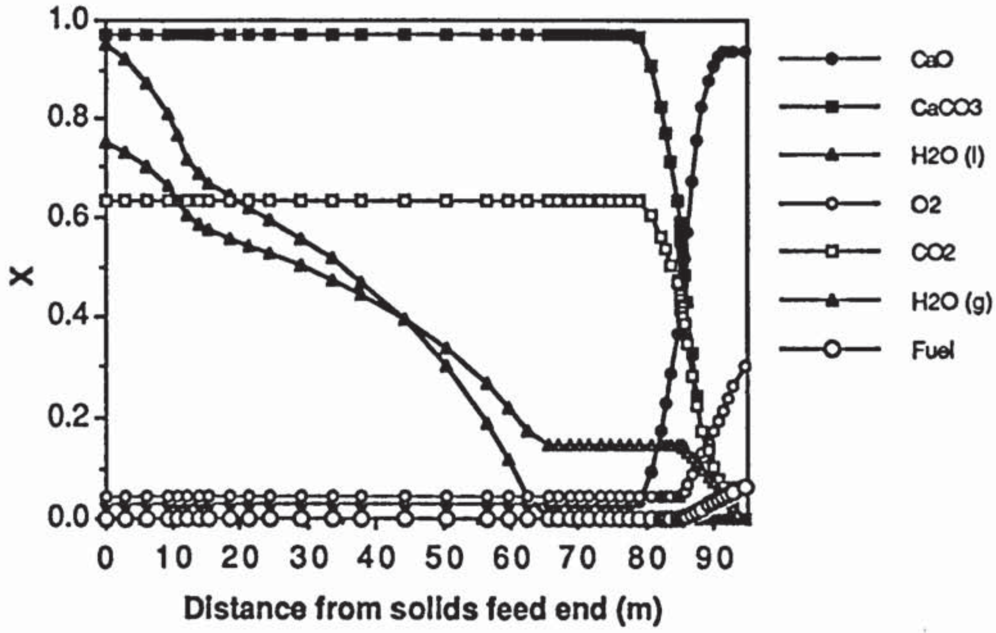


Figure 7.15: Variation of concentration of solid and gas phases in kiln with distance from solids feed end.
X_i = Mass Fraction of Compound i; see Appendix A for more details.

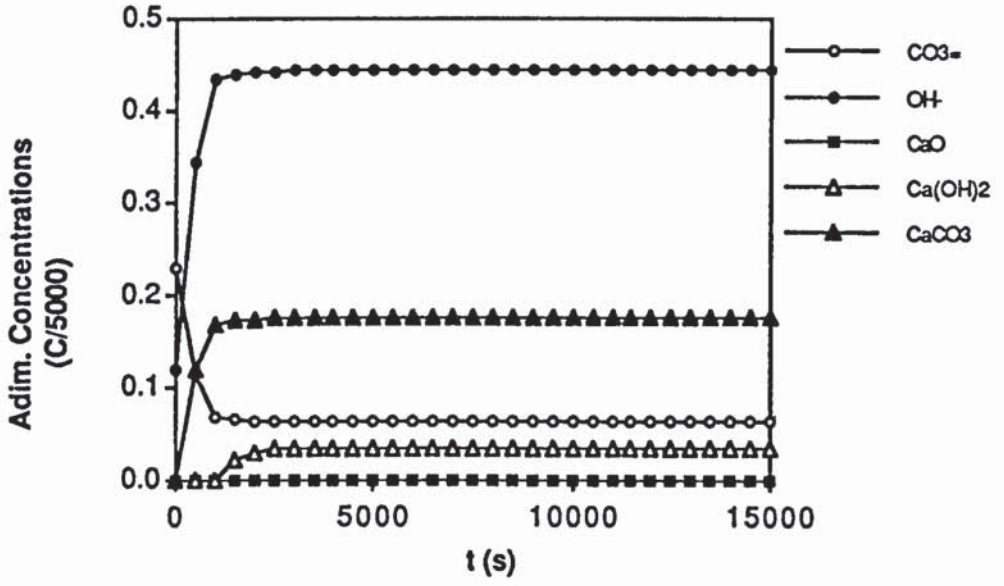


Figure 7.16: Evolution of concentrations in the slaker during start-up. Adimensionalization constant is 5000 mol/m^3.

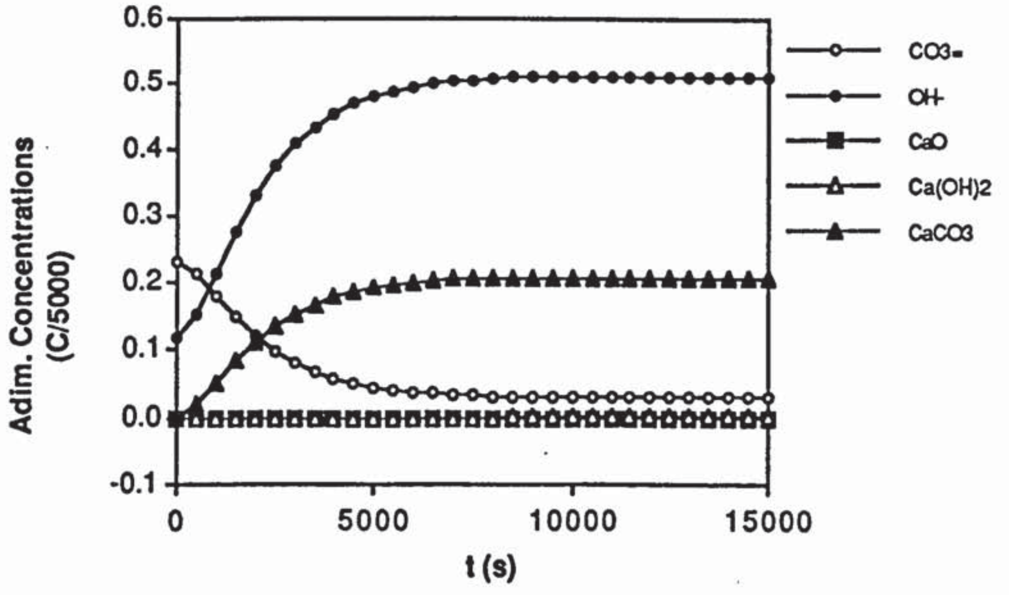


Figure 7.17: Evolution of concentrations in the first causticizer during start-up. Adimensionalization constant is 5000 mol/m^3.

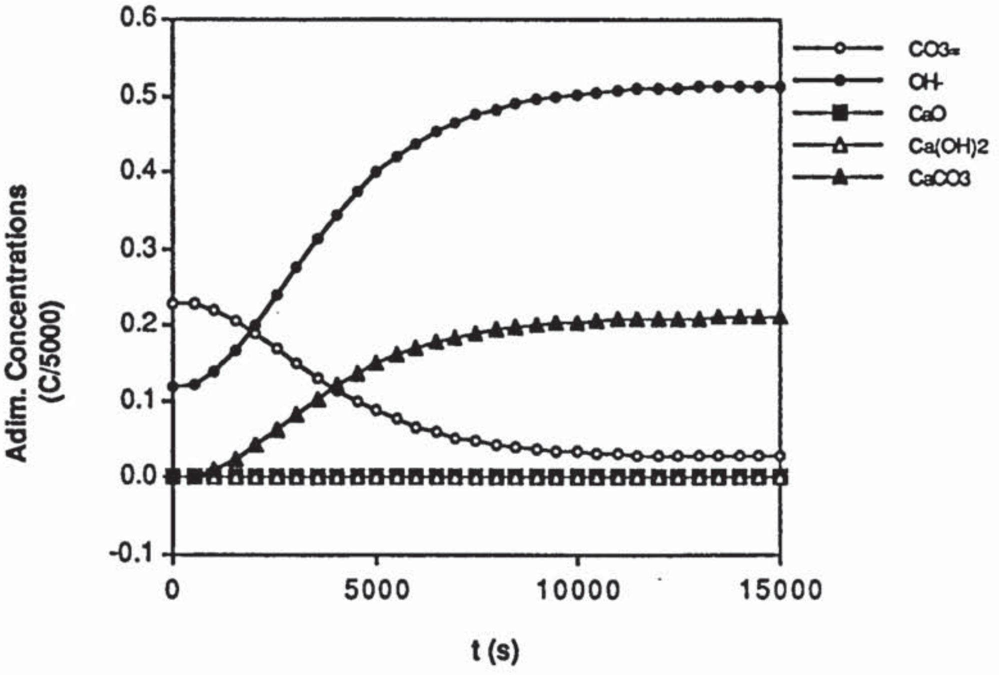


Figure 7.18: Evolution of concentrations in the second causticizer during start-up.
Adimensionalization constant is 5000 mol/m^3.

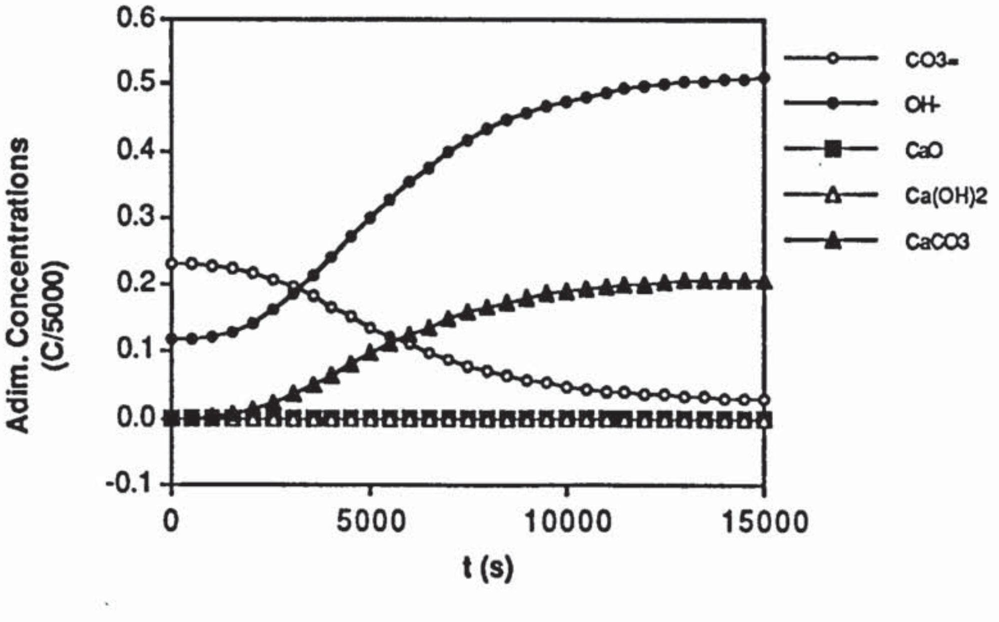


Figure 7.19: Evolution of concentrations in the third causticizer during start-up.
Adimensionalization constant is 5000 mol/m^3.

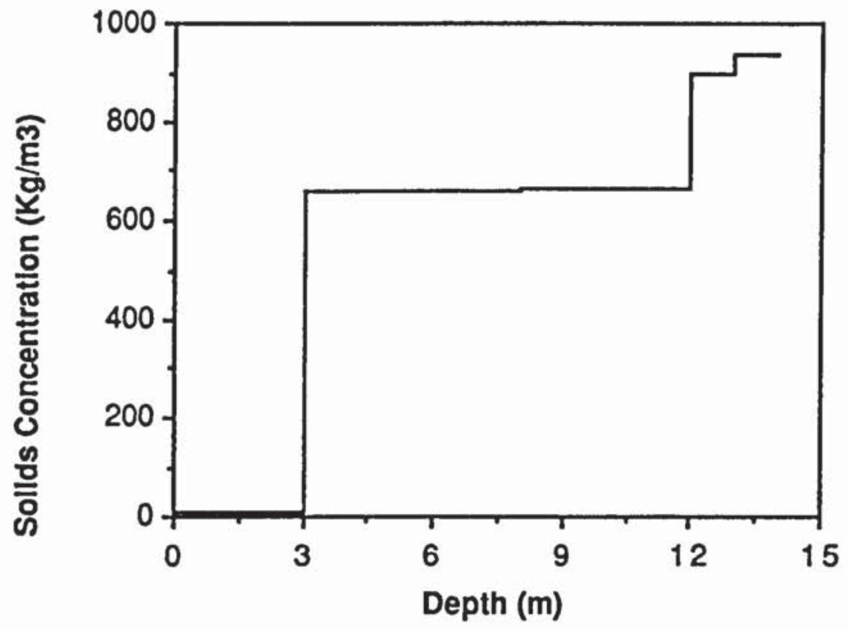


Figure 7.20: Typical variation of solids concentration with depth in the white liquor clarifier.
Depth = 0 corresponds to the top of the clarifier.

7.5.2 Effect of Decomposition of the Overall Model

Since the mathematical model for the physical process has been decomposed into several submodels which interact only at discrete moments in time, rather than continuously, the effect that this approach may have on the *accuracy* of the solution obtained must be investigated. The overall model, which was initially described by a continuous time-continuous state pattern (see Chapter 5), has been replaced by an approximate model, which is continuous time-discrete event. The state of each individual model still varies continuously with time, but the events that trigger these changes are allowed to happen only at discrete times (when the interactions are established, or when the user changes some parameter).

As a practical example, let us consider the case where the concentration of the liquor from the green liquor storage, which is fed to the slaker, suffers a continuous variation in time, given by $C_{CO_3^{2-}} = 1150 + 200sin(0.02t)mol/m^3$. One of the input variables to the slaker is the concentration of this stream. Updating the interactions at certain time intervals only, rather than continuously,

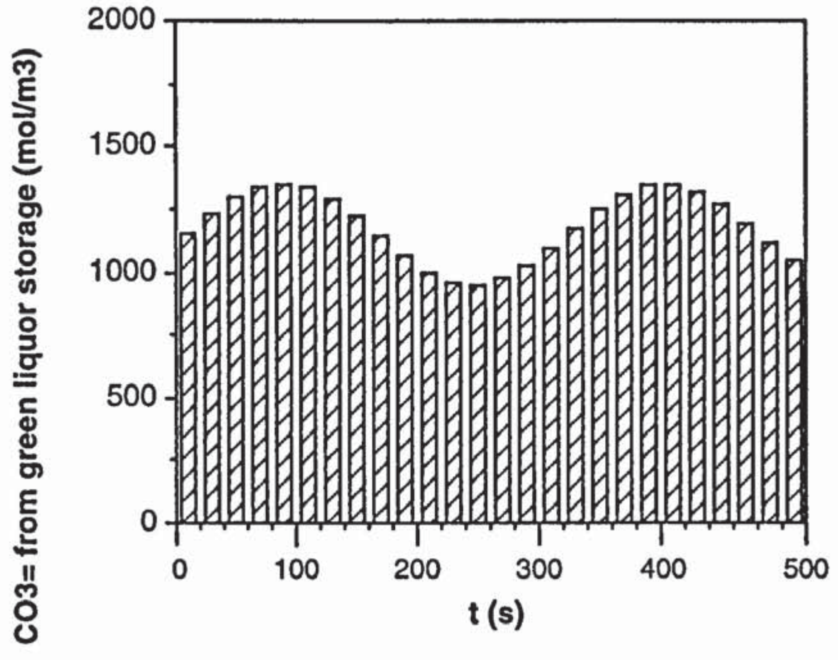


Figure 7.21: CO_3^{2-} concentration from green liquor storage: update every 20s.

corresponds to keeping the inputs to a unit constant during that interval, and updating them only at the end.

If the interactions between the green liquor storage tank and the slaker are reset every 20 s, the input to the slaker is represented in Fig.7.21. The time interval is short enough, in this case, to capture the process dynamics, i.e., Fig.7.21 offers a good approximation to the real input (sine wave). If the interactions are reset every 100 s, which is a much coarser approximation, the input to the slaker is represented in Fig.7.22. This time interval is obviously too long and leads to a corresponding degradation in the quality of the solution (see Fig.7.23). However, the response of the concentration in the slaker is very close for time intervals of 10 and 20 s, and can be taken as an acceptable approximation in these cases. The maximum value of the acceptable time interval after which the interactions must be reset is obviously problem-dependent, and some kind of study must be performed in order to ensure that the dynamics of the real system are being correctly captured.

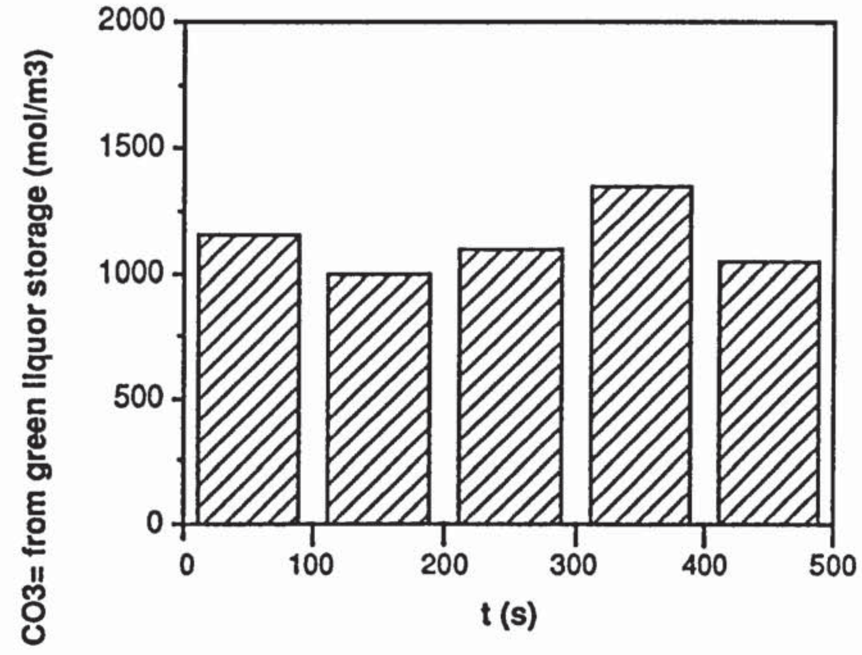


Figure 7.22: CO_3^{2-} concentration from green liquor storage: update every 100s.

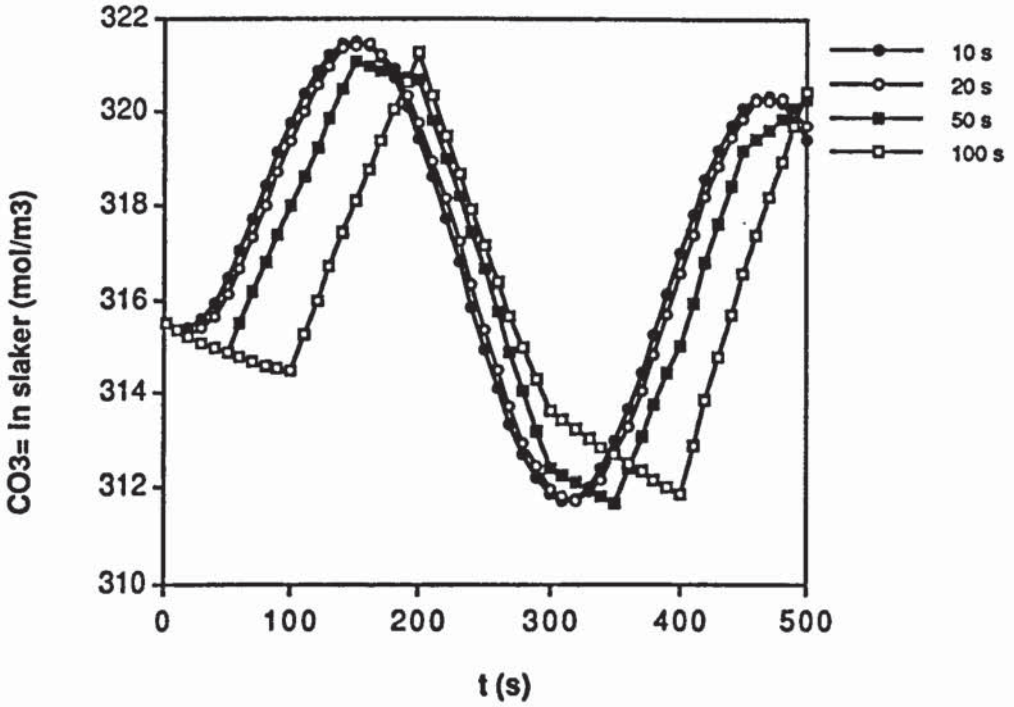


Figure 7.23: CO_3^{2-} concentration in slaker: effect of different interaction intervals.

7.5.3 Effect of Concurrent Execution

Another important factor is the extension of the benefits obtained by distributing the numerically-intensive calculations throughout the network, i.e., the advantages it brings in terms of *execution speed.* In the configuration of the system that corresponds to the chemical recovery cycle (i.e., one lime kiln, one causticizing battery, one white liquor clarifier and one mud filter), the execution time is totally dominated by the integration of the equations that describe the lime kiln, since it can take up to, and in some cases more (depending on the initial conditions), than 20 minutes to perform the first integration (from $t = 0s$ to $t = 10s$). An obvious conclusion is that the model for this unit is too heavy, in terms of computational speed, to be used interactively. In this case, in order to be effective, the use of parallel processing techniques must be done at the *algorithm level*, using a parallelized algorithm to integrate stiff differential equations, as well as at the *object* level.

In a general case, for a coarse decomposition (i.e., few model objects, each described by a computationally demanding model) the difference between one and more available processors will not be very large, since execution time will in any case be close to the time it takes to update the "limiting", i.e., the slowest, unit. This is a consequence of inserting procedural routines in an object-oriented environment — it is no longer possible to deal only with lightweight objects, whose behaviour is described by simple and short functions.

In order to clearly determine the effect of the availability of several processors to execute, concurrently, the numerical calculations associated to the updating step, the execution time was measured for systems composed by one up to ten causticizing batteries (see Fig.7.24). Note that here *execution time* is not being employed in the sense of CPU time, but rather as the real time elapsed between the start and the end of the updating process, as determined by calls to the Unix system function `date` at these moments. Each point corresponds to the mean of three measurements. Note that this data are intended as a qualitative measure only. Two main factors influence the performance of the distributed version, namely the number of processes executing in the machines where the remote processes are spawned, and the time necessary to send and receive the

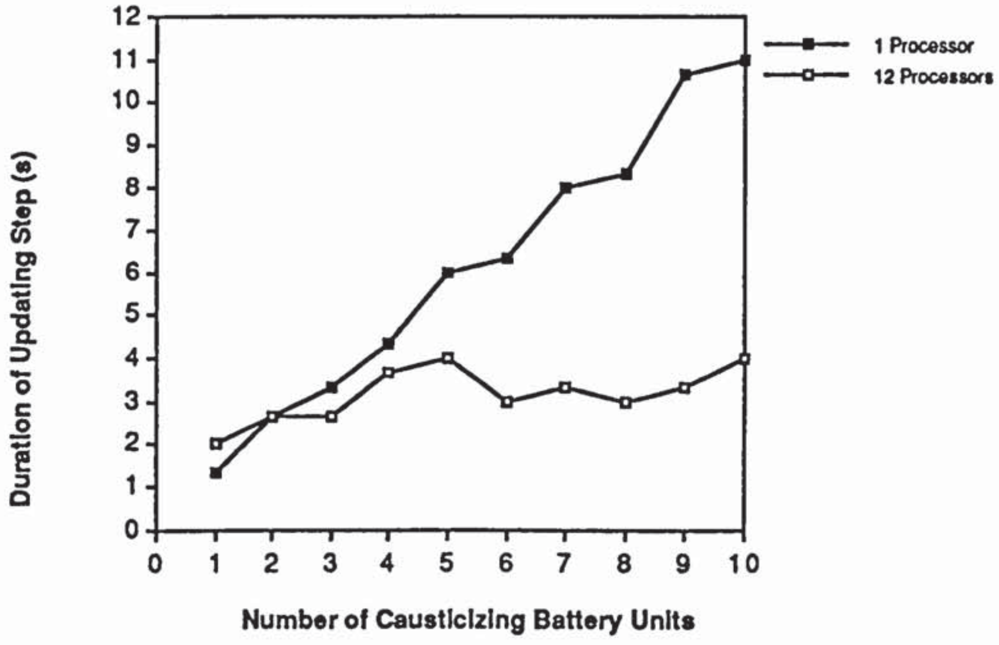


Figure 7.24: Duration of the updating step (from $t = 0s$ to $t = 10s$) as a function of the number of causticizing batteries.
Each processor corresponds to a different Sun Classic workstation, model 4/15N-16-P40.

messages with the data, which is strongly dependent on network traffic. In order to determine the quantitative influence of both these factors, a deeper study would have to be performed and is out of the scope of this work. However, it is obvious from Fig.7.24 that in the case of a single available processor the time required for the first updating step is additive, i.e., directly proportional to the number of causticizing battery units of the system (which is to be expected, since the calculations are executed sequentially). In the case of twelve available processors, for one and two causticizing battery units, the time is either equal or higher than the time required in the case of a single processor, but then increases only slightly. This is also to be expected since, for one causticizing battery, the distributed version involves an additional step, which is to send and receive messages. This overhead rapidly becomes negligible as the number of units increases and the effect of updating them concurrently outweighs the time required to send and receive the messages.

In conclusion, the standard classes developed in this work provide a framework into which different types of model, view or controller objects can be inserted. The range of applicability of this software system is obviously strongly dependent on the number and diversity of the model classes available; as long as new classes are subclassed from existing ones, they will be compatible with the framework. An example of the application of the prototype to the simulation of an industrial system is presented. The approach proposed has enabled the interactive simulation of a set of very complex simulation modules, thus offering not only visual feedback to the user but also a simpler way of defining the configuration of the process model and access the parameters used in the simulation.

Chapter 8

Conclusions and Future Work

8.1 Introduction

As mentioned in Chapter 5, section 5.8.2, the initial aim of this project was to develop a simulation package for the chemical recovery cycle of a paper pulp mill. The package should include both the implementation of detailed mathematical models for the main industrial units involved in the cycle, and a graphical user interface to enable on-line interaction with the simulation modules. The mathematical models were not available to the required degree of detail. Possibly due to the limitations, in those days, on computational power, previous work on the simulation of the cycle as a whole used very simple models (Jacobi and Williams, 1973a,b).

The first step was therefore to develop (either by adapting published mathematical models for similar units, or by developing entirely new models) and implement a series of Fortran77 subroutines which simulated the behaviour of the main units. Although unsuitable for a number of applications, especially those in which dynamic allocation of memory must be performed, Fortran77 is still a powerful tool for the implementation of procedures where intensive use is made of numerical calculations. This is due to the fact that the language itself was designed for that purpose, i.e., to enable a straightforward translation of equations into the programming language (Fortran stands for FORmula TRANslation) and also because of the wealth of standard, freely available implementations of currently used numerical methods. The simulation of the lime kiln, for example, required the solution of a set of hyperbolic partial differential equations, in space and time. A possible solution of the model consists of discretizing the spatial derivatives using finite differences, and integrating the resulting system of ODE's in time. A better approximation to the spatial profile is however obtained if a piecewise polynomial approximation is used, i.e., the spatial domain is divided into finite elements and a different third degree (Hermite) polynomial is used to approximate the profile in each of them. The resulting system of ODE's can then be integrated in time. Just for this module, two different groups of standard Fortran77 software were used, namely the LSODI integrator (Hindmarsh, 1980) for stiff differential equations and the orthogonal collocation routines developed by Villadsen and Michelsen (1978).

After some of these routines had been implemented, i.e., a very large amount of result data was being produced, work started on the first version of the prototype interface. At first, no consideration was given to the need to allow for changes in the physical system; its configuration was taken to be a typical one, i.e., one which prevailed in a number of paper pulp mills. X Window, as a windowing system, offered a great number of advantages, as mentioned before. Although not impossible, it is very complicated — and ineffective — to use plain X Window, without taking advantage of available higher-level software (toolkits), for the development of complex GUI's. Since Sun workstations are widely spread in the scientific community, the XView toolkit, which follows the OpenLook guidelines (Hoeber, 1988) and has become a Sun standard, was used. Although this may somewhat reduce the portability of the resulting software (especially for those who advocate Motif as *the* look), no differences in performance have been reported between the two. At first, only the Xlib primitives and XView toolkit were used to generate the GUI, i.e., no framework was provided for the application. The first version of the GUI was implemented in C.

Several problems soon became apparent. The size of the application made it difficult, without using any type of framework, to know which event handler(s) (non-bound C procedure) had been registered with each widget, and then to determine which widget(s) was related to each simulation module. This occurred in spite of a careful division of functions among files. Also, the system was not flexible since any change in the configuration of the physical system would require source code re-writing, and a careful determination of all the repercussions this might have throughout the rest of the application. Although one of the functions of graphical user interfaces is to hide the complexity of the underlying system from the user, and this objective was attained, this clarity should not be obtained at the expense of a badly structured application code which suffers from poor modularity and re-usability. As an example, none of the types of display provided would be easily re-used since they had all been tailor-made for this specific application.

At this point, the need for a different approach became obvious. If the interface was to be applicable to a wider range of problems, a framework and a greater degree of flexibility would have to be provided. The main objective of the work shifted from the development of a specific simulation package to that of a general-use simulation framework, which should provide *hooks* that allowed application developers to insert user-defined simulation modules and to define the types of display and interaction adequate for their specific problem. This shift represents the first conclusion that can be drawn from this work: *toolkits per se are not enough to develop a structured application.* They provide pre-defined sets of interactive objects, usually created by a simple function call, thus significantly reducing the amount of code to be written by the application developer; they also provide a consistent widget appearance and interaction behaviour (e.g., how the appearance of a button changes when it is pressed) throughout the application. However, they do not separate the interface from the problem and, if a large number of widgets is needed, the existence of a framework in which to insert them in an organized way becomes essential.

Since a framework can be defined as a group of interdependent objects which cooperate to accomplish a certain task (this corresponds to a *behavioural composition*, as mentioned in Chapter 2), the use of object-oriented concepts is essential for the design of the framework. C++, according to some the most widely spread object-oriented language in and out of academic environments, was selected as a convenient means to implement the framework. It also provided the possibility of continuing to use the X and XView C-interface function calls.

The integration of the simulation modules for the chemical recovery cycle into the framework would be a good test of the effectiveness of the approach since it represents the migration of a set of complex, non-standard routines written in a procedural language into an object-oriented graphical interactive environment. A working prototype which exemplifies the application of the framework to this problem was therefore developed and has been described already in previous chapters. The main conclusions that can be drawn from this work are presented in the next section.

8.2 Conclusions

At present, *there is a gap between new emerging concepts such as object-orientation and graphical user interfaces, and the traditional approach to complex simulations, i.e., batch runs of long procedural codes.* The object-oriented approach was developed and is especially dedicated to systems that do not involve the use of complex numerical algorithms in order to determine the state of each object. Although the overall system may be quite complex, and composed of many objects, the behaviour of *each* of these is assumed to be fairly simple and described by short functions; typical examples of object-oriented concepts include graphical user interfaces and library and information systems. For systems like an industrial unit, or a solid catalyst particle undergoing chemical reaction, which are normally described by sets of equations (often partial differential equations), the notion of object becomes somewhat blurred. Two approaches are possible to simulate these systems in an object-oriented environment: either to treat them as high-level objects, whose behaviour is described by complex sets of equations, or to decompose them into simpler objects. In this last case, the decomposition must usually be made according to something which is not directly related to the physical problem domain — the mathematical algorithm used for the solution of the model. This will determine the class hierarchies required and can give rise to entities like *equation*, *finite element*, *partial derivative*, *time increment*, etc.. From this, it follows that a purely object-oriented simulation of such systems depends primarily on the development of OO numerical algorithms. In this work, the first approach was used, i.e., procedural simulation modules were imported into an object-oriented environment and "wrapped" by methods; although this is not a strictly OO approach, it represents a way of re-using existing software. Moreover, the efficacy of OO concepts for numerically intensive applications is still under study, as it involves a higher number of function calls (or similar, depending on the implementation language) than its procedural counterpart, which can represent an unsurmountable time overhead. If OO proves ineffective for some numerical methods, the insertion of procedural pieces of code may be the only way to port this kind of simulation into a graphical, interactive environment.

Secondly, *the main deficiency of existing simulators is lack of flexibility; this can be avoided if a library of simulation classes is developed, rather than a specific simulator.* Most simulators offer simulating facilities for a specific problem domain only, although some of them are now turning to the simulation of mixed domain areas (Zobel and Lee, 1992). For a specific domain, the configuration of the system can be defined on-line, and in most cases user-defined simulation modules added. However, the interface presented by the *simulator* itself is not adaptable, nor are the types of display available; i.e., each simulator offers a specified and fixed set of functionality and presentation options. As stated by van der Meulen (1987) (sic), "most simulation systems are closed, i.e., there is no access to the source code. Even if they were open, one would have to read and modify a lot of source code". This work aims to increase the flexibility of such software by developing a set of simulation classes, which can be re-defined and extended at any level, i.e., at the problem domain, presentation or interaction levels. Toolkits represent a higher level than the graphics/windowing systems they are based upon. Although they may be sufficient, *per se*, for small applications which have few windows (e.g., the XView interfaces for the Internet Newsgroups, the mail tool, or the debugger), large applications become a tangled mess of procedures, as mentioned in the previous section. In order to create easily new types of displays, where a single window may possess any type and number of widgets, and, most importantly, to know which presentation and interaction objects are related to each information model, an even higher level is needed. A possible approach is to create an object-oriented framework in which the widgets are grouped to form higher level objects, such as the *views* developed in this work. The event handlers registered with all the widgets related to a view can then be grouped as methods of another object: the *controller* of that view. This has greatly simplified the application code itself, as a complex view is generated simply by dynamically creating a view object of the desired class. It also immediately establishes which widget(s) are related to which part of the problem, since each view-controller pair is related to one *model* object only; therefore, if a view is registered as a dependent of a model, all its widgets relate to that model. *The definition of views as high-level objects which group widgets has made the use of the toolkit considerably easier.*

As stated above, the strength of this approach is the possibility of refining existing classes, in order to develop a simulator which is ideally suited for a specific problem. For example, the main window of the simulator is, in the prototype developed, an object of class Draw_View. New subclasses can be created in which the functionality of this class is extended or re-defined, thus effectively changing the way in which the model for the pysical system is defined on-line. Such a degree of freedom is especially important in the case of non-standard user-written simulation modules, which are normally not present in available simulators, and for which any type of display, from line and bar charts to contour charts, may also be required. The existence of an integrated environment, which includes editors and class hierarchy and object browsers, would allow the application developer to introduce those changes to the system on-line. Such environments have been developed in Smalltalk (Goldberg, 1984) and KEE (Fikes and Kehler, 1985; Filman, 1987). KEE also supplies two different types of graphics packages, namely one that contains pre-determined images such as gauges and dials, and one which which consists of even more basic components such as lines, circles, etc. (Kempf *et al.*, 1987). C++ does not provide graphics capabilities by default, although they can be introduced as class libraries (as was done in this work). Moreover, no generally available C++ environment supports the run-time creation and integration of new types into running processes, although this is possible (Jordan, 1990). Although this means that the changes to the class library have to be made off-line, using an editor, the absence of these features in the C++ language allows it to be used in a wider set of system environments than other object-oriented languages (Jordan, 1990). The introduction of such an environment has another detrimental effect in this specific case, in which graphical user interfaces composed by several displays each with a possibly large number of widgets are likely to be created: it consumes computer power, thus effectively reducing the amount which can be dedicated to the application itself. *The advantages gained by providing a fully integrated environment must be weighed against its consumption of computer resources, especially when the application itself is likely to be computationally heavy.* Most simulations of real systems fall in this category and both the interface and the entities associated with the problem domain may need these resources. A situation which occurred several times during trial runs of of the prototype was the impossibility of debugging it using the debugger due to memory limitations.

In this work, an MVC (Model-View-Controller) framework was developed, which is specifically prepared for the existence of several (possibly many) submodels, views and controllers. Although the functions performed by the objects are similar to the ones provided by Smalltalk, no attempt was made to reproduce either its class library or syntax. A "triangular" framework such as MVC makes it possible to update the view objects without intervention from the respective controllers. This makes it easier to insert adaptors between the model and the view objects then it would be if the PAC (Presentation-Abstraction-Control) framework (Coutaz, 1987) were used, since the Control components stands between the Presentation and Abstraction components. Except for this implementation detail, *any other framework can be used, as long as the separation between the problem domain and the presentation and interaction levels is retained.* The framework developed in this work is a very light one, in which the task of the objects is fundamentally to relate in an organized manner data entities which would be present in any case; i.e., the framework itself does not impose a heavy additional burden. Most of the frameworks referred to in Chapter 3 could be used to perform the same functions.

Although the base class library developed consists of only the base classes needed for this specific application, the number of classes related to the *view* hierarchy is quite large. Some of these classes provide an immediate way to create views associated to operations that occur very frequently such as loading or saving a file (Load_File_View and Save_File_View), input a string (String_View) choosing a font (Fonts_View) or a colour (Palette_View). Once again, *the definition of views as high-level objects promotes re-usability and ease of use.*

The simulation of models which are computationally demanding raises the need for concurrency/parallel processing techniques. In order to conjugate concurrency and objects, actor languages immediately come to mind. However, actor languages are very low-level (Booch, 1991) and difficult to use; their applicability to problems where the algorithms needed are themselves very complex and where a high degree of sophistication is required for the GUI is doubtful. As a carefully designed hybrid, which has profited from a considerable amount of practical experience, C++ offers both the advantages of an object-oriented language and an easy access to procedural languages such as C or Fortran; most importantly, it does so while retaining all the execution efficiency provided by C. It does not, *per se*, offer concurrency features. The development of software libraries such as PVM, which is still evolving at this date, offers two major advantages. The first one is that no limitations are imposed on the type of concurrency paradigm used. At a first level of distribution, the framework developed in this work provides the possibility of distributing the updating methods for each object throughout the network. However the user is free to use either sequential or parallelized *algorithms*; any of these are perfectly compatible with the framework. The spawned process that performs the evaluation of the state of a model may spawn other processes itself, each of which performs a certain share of the necessary calculations. This is possible since PVM does not impose any type of paradigm, i.e., a spawned process is at the same level as the process that created it. Secondly, because concurrency is introduced in the form of libraries (which is the current trend for developing concurrent software), no changes were made to the standard C++ language and the standard C++ compiler for Sun Sparc workstations was used. *Introduction of concurrency by means of libraries such as PVM, rather than by modifying the language and therefore having to use a modified compiler, provides a greater degree of portability without imposing* *restrictions on the type of paradigm used.*

The implementation of computationally heavy algorithms as separate processes, on other machines available in the network, offers another advantage which is not immediately obvious: *it distributes the memory load* thus avoiding the occurrence of out-of-memory situations in the local host. Since the graphical user interface itself consumes a great amount of memory, this situation may occur in cases where the objects need, for their updating functions, to store large scale arrays and vectors, which is not uncommon in cases where implicit methods are used for the integration of differential equations. The distributed approach taken in this work effectively allows the local host (specifically, the host where the application is running, not necessarily the one where the display of the results is performed) to be relieved from the number-crunching tasks.

The implementation of this software system was an arduous task, mostly due to the number of languages and libraries used (C++, Fortran77, X Window and XView, PVM, standard Fortran77 routines of numerical methods) and the need to interface them correctly. Moreover, as mentioned before, X and X-based toolkits provide little support for graphics. However, in addition to greater difficulties during the software implementation stage itself, development of an application of this dimension *without* a well-defined framework would have resulted in, at best, a monolithic, virtually non-reusable application. *Interactive simulation of real systems is inherently complex; the methodology proposed in this work has enabled the construction of a highly structured and very flexible prototype.*

8.3 Future Work

As mentioned above, plain C++ does not provide editing/browsing tools. This means that the application developer (who may well be the end-user) is not able to generate new sets of derived classes on-line, or to access a graphical representation of the class hierarchies. Although the existence of a fully integrated environment also has disadvantages, as pointed out in the previous section, some light-weight facilities, such as a class browser, would help the developer visualize the existing class hierarchies and could greatly increase the ease with which the framework can be applied to different problems.

The problem-domain hierarchies developed in this work are very basic, since the purpose was to develop the framework rather than a problem-specific set of model libraries. As a consequence, the model classes used were subclassed directly from the top model classes *unit* and *stream*. For effective re-use, specialization must be progressive, allowing incremental refinement of the model characteristics until the desired behaviour is attained. If application of the framework to a specific problem domain is desired, then full model class hierarchies, enough to simulate most of the physical systems in that area, must be available. Note however that if the simulation of non-standard physical systems is desired, it is still possible to subclass any such model class directly from the base classes at the top of the standard hierarchy.

A general event recovery framework, such as the one proposed by Wang and Green (1991), would be an important asset. In this specific case, it could applied, among other things, to the retrieval of the state of the models for previous values of time and combinations of parameters.

As mentioned in Chapter 7, the lime kiln takes a considerably longer time to update itself than all the other units. When this situation occurs, and if the response time of a certain unit is so long that it jeopardizes the feasibility of an interactive approach, then a parallelized algorithm must be used for that unit. The development of such algorithms is primarily of a mathematical nature and out of the scope of this work. However, if models are high-level objects, this need is likely to arise in most simulations, and the possibility of using a distributed algorithm for the integration of differential equations would be a valuable addition to the system.

Other than the possibilities for improvement mentioned above, it would be highly desirable for the application to become totally independent from Fortran source code. Although at present it seems to be the only solution, the insertion of calls to Fortran subroutines makes it necessary to use both the C++ and Fortran77 compilers. Moreover, the C++ (or C)/Fortran 77 interface is compiler-dependent, which decreases the degree of portability of the application. C++ is equally well-suited both for graphical user interfaces and, due to its performance capabilities, to numerical calculations. The fact it was used as the main programming language will enable progressive replacement of procedural routines by object-oriented classes that implement numerical methods (if these prove to be feasible in practice).

Two levels of usage may be associated with the prototype developed. One of them is the end user, who need not know the internal structure of the application, but for whom the GUI must provide a simpler and clearer way to interact with complex simulations, whilst offering the advantages of visual feedback. An example of such a user is the operator of an industrial plant, for whom this type of software represents a learning tool, providing a safe way of getting acquainted with and learning how to interact with the industrial system itself. An example of a different type of user is a process engineer or designer, who may wish to adapt the tool for teaching purposes (e.g., to train the staff as mentioned above) or to study the effect of different new industrial configurations or equipment. This work is aimed at this last type of user, who will need to know the standard class library in order to be able to extend it in the desired way.

This study has tried to bridge the gap between the traditional approach to complex simulations and new concepts such as object-orientation and graphical user interfaces. The work developed shows that the application of computer science concepts to the engineering domain can yield considerable benefits by helping domain experts to tailor interactive tools to suit their needs.

References

Achauer, B. (1993). The DOWL Distributed Object-oriented Language. Communications of the ACM **36**:9, 48–55.

Adams, T. N. (1992). Lime Reburning. 1992 Kraft Recovery Operations Short Course/Tappi Notes, 59–72.

Agha, G. (1986). Actors: a Model of Concurrent Computation in Distributed Systems. Cambridge, Massachussets, USA: MIT Press.

Agha, G. (1989). Foundational Issues in Concurrent Computing. SIGPLAN Notices **24**:4, 59–65.

Agha, G. (1990). Concurrent Object-oriented Programming. Communications of the ACM **33**:9, 125–141.

Alexander, H. (1987). *Formally-Based Tools and Techniques for Human-Computer Dialogues.* Chichester, UK: Ellis Horwood Limited.

Alkema, K. L. (1971). The Effect of Settler Dynamics on the Activated Sludge Process. MSc Dissertation, University of Colorado, USA.

Angevine, P. A. (1983). Cost-effective Causticizing. 1983 Engineering Conference/TAPPI Proceedings, 253–265.

Apple Human Interface Guidelines: The Apple Desktop Interface. Reading, Massachussets, USA: Addison-Wesley.

Armstrong, M. P., Densham, P. J. (1992). Domain Decomposition for Parallel Processing of Spatial Problems. Computer Environment and Urban Systems **16**, 497–513.

ASPEN (1988). Model Manager^{TM} Automates the Use of Rigorous Process Models. The ASPEN Leaf **5**:2, 6–7.

Ataíde, J. (1990). Balanco Mássico ao Forno da Cal - Documentation supplied by Soporcel, Leirosa, Portugal (In Portuguese).

Athas, W. C. and Boden, N. J. (1989). Cantor: An Actor Programming System for Scientific Computing. SIGPLAN Notices **24**:4, 66–68.

Attir, U., Denn, M. and Petty, C. A. (1976). Dynamic Simulation of Continuous Sedimentation. AIChE Symposium Series **73**, 49–54.

Ayers, K. E. (1990). The MVC Paradigm in Smalltalk-V. Dr. Dobb's Journal, November 1990, 168–175.

Cheremisinoff, N. P. and Azbel, D. S. (1983). *Liquid Filtration.* Woburn, Massachussets, USA: Ann Arbor Science Publishers.

Bailey, R. B. and Willison, T. R. (1985). Computerization of the Recovery Cycle — Lime Kiln Control and Optimization. 1985 Pulping Conference/Tappi Proceedings, Hollywood, 619-628.

Bain, W. L. (1989). Indexed, Global Objects for Distributed Memory Parallel Architectures. SIGPLAN Notices **24**:4, 95–98.

Bär, M. and Zeitz, M. (1990). A Knowledge-based Flowsheet-oriented User Interface for a Dynamic Process Simulator. Computers and Chemical Engineering **14**, 1275–1289.

Barfield, L. (1993). *The User Interface, Concepts and Design.* Wokingham, England: Addison-Wesley Publishing Company.

Barth, P. S. (1986). An Object-Oriented Approach to Graphical Interfaces. ACM Transactions on Graphics **5**:2, 142–172.

Bass, L., Faneuf, R., Little, R., Mayer, N., Pellegrino, B., Reed, S., Seacord, R., Sheppard, S. and Szczur, M. (1992). A Metamodel for the Runtime Architecture of an Interactive System. SIGCHI Bulletin **24**, 32–37.

Bensley, E. H., Brando, T. J., Fohlin, J. C., Prelle, M. J. and Wollrath, A. M. (1989). MITRE's Future Generation Computer Architectures Program. SIGPLAN Notices **24**:4, 99–101.

Birrell, A. and Nelson, B. (1984). Implementing Remote Procedure Calls. ACM Transactions on Computer Systems **2**:1, 39–59.

Black, A., Hutchinson, N., Jul, E., Levy, H. and Carter, L. (1986). Distribution and Abstract Types in Emerald. IEEE Transactions on Software Engineering **13**:1, 39–59.

Blackwell, B. R. (1987). Increasing White Liquor Causticity by Addressing the Diffusion Limitation. Pulp and Paper Canada **88**:6, 87–94.

Blevins, T. and Rice, R. (1983). Automating a Lime Kiln Control. Tappi Journal **66**:3, 103–105.

Booch, G. (1991). *Object-oriented Design with Applications.* Redwood City, Calfornia, USA: The Benjamin/Cummings Publishing Company.

Borland International (1990). Turbo C++ Pack and Manual.

Borning, A. H. and Ingalls, D. H. (1982). Multiple Inheritance in Smalltalk-80. Proc. Nat. Conf. Artificial Intelligence, Pittsburgh, Pensylvannia, USA, 234–237.

Bösser, T., Melchior, E.-M. (1992). The Sane Toolkit for Cognitive Modelling and User-Centred Design, in *Methods and Tools in a User-Centred Design for Information Technology.* M. Galer, S. Harker and J. Ziegler (Eds.), North-Holland: Elsevier Science Publishers B. V.

Bösser, T. (1994). Models of User Behaviour and Measures of Usability. In *Human-Machine Communication for Educational Systems Design.* M. Brouwer-Janse and T. Harrington (Eds.), Berlin: Springer-Verlag.

Bourne, J. R. (1992). *Object Oriented Engineering - Building Engineering Systems Using Smalltalk-80.* Boston, USA: R. D. Irwin, Inc. and Aksen Associates, Inc..

Briot, J. P. (1989). From Objects to Actors: Study of a Limited Symbiosis in Smalltalk. SIGPLAN Notices **24**:4, 69–72.

Briot, J.-P. and Cointe, P. (1989). Programming with Explicit Metaclasses in Smalltalk-80. Proc. OOPSLA'89, 419–431.

Brooke, J. (1993). A Flexible User Interface Toolkit for Building Adaptable Systems. Proc. Interacting with Images, London, UK, 1–14.

Burbeck, S. (1987). Applications Programming in Smalltalk–80: How to Use the Model-View-Controller (MVC) Evaluation. Publication of SoftSmarts, Inc., 1–23.

Bryce, J. R. G. (1980). *Pulp and Paper: Chemistry and Chemical Technology*, Vol.I (Third Edition). James P. Casey (Ed.), New York, USA: John Wiley and Sons.

Campbell, A. J. (1981). Factors Affecting White Liquor Quality: Green Liquor Concentration, Dregs Concentration and Lime Dosage. Pulp and Paper Canada **82**, 78–87.

Campbell, R. H., Islam, N., Raila, D. and Madany, P. (1993). Designing and Implementing CHOICES: An Object-oriented System in C++. Communications of the ACM **36**:9, 117–126.

Card, S. K., Moran, T. and Newell, A. (1983). *The Psychology of Human-Computer Interaction.* Hillsdale, New Jersey, USA: Lawrence Erlbaum.

Carey, G. (1994). A Prototype Scalable, Object-oriented Finite Element Solver on Multicomputers. Journal of Parallel and Distributed Computing **20**, 357–379.

Caromel, D. (1989). A General Model for Concurrent and Distributed Object-oriented Programming. SIGPLAN Notices **24**:4, 102–104.

Caromel, D. (1993). Toward a Method of Object-oriented Concurrent Programming. Communications of the ACM **36**:9, 90–102.

Carr, H. H. (1992). Factors that Affect User-friendliness in Interactive Computer Programs. Information and Management **22**, 137-149.

Carré, F. and Cléré, P. (1989). Object-oriented Languages and Actors: Which Language for a Distributed Approach? SIGPLAN Notices **24**:4, 73–75.

Carroll, J. M., and Rosson, M. B. (1991). Getting Around the Task-artifact Cycle: How to Make Claims and Design by Scenario. Research Report IBM RC 17908, 1–42.

Caustificação e Forno da Cal — Documentation supplied by Soporcel, Leirosa, Portugal (In Portuguese).

Charos, G. N., Arkun, Y. and Taylor, R. A. (1991). Model Predictive Control of an Industrial Lime Kiln. Tappi Journal **74**:2, 203–211.

Chase, J., Amador, F., Lazowska, E., Levy, H. and Littlefield, R. (1989). The Amber System: Parallel Programming on a Net of Multiprocessors. Proceedings of the 12th ACM Symposium on Operating System Principles. New York, USA: ACM.

Chi, J.-C. (1974). Optimal Control Policies for the Activated Sludge Process. PhD Dissertation, State University of New York at Buffalo, USA.

Chignell, M. H. and Waterworth, J. A. (1991). WIMPS and NERDS: An Extended View of the User Interface. SIGCHI Bulletin **23**, 15–21.

Christopher, T. (1989). Message Driven Computing and its Relationship to Actors. SIGPLAN Notices **24**:4, 76–78.

Ciccarelli, E. C. (1984). Presentation Based User Interfaces. Technical Report 794, Artificial Intelligence Laboratory, Massachussets Intelligence Laboratory, USA.

Coad, P. (1991). Adding to OOA Results. Journal of Object-oriented Programming, May 1991, 64–69.

Coad, P. and Yourdon, E. (1991). *Object-oriented Analysis* (Second Edition). Englewood Cliffs, New Jersey: Prentice-Hall.

Cointe, P. (1987). Metaclasses are First Class : the ObjVlisp Model. Proc. OOPSLA'87. Special Issue of SIGPLAN Notices **22**:12, 156–167.

Colbrook, A. (1993). Parallel Processing: a Practical Proposition? IEE Review **39**, 119–121.

Comings, E. W. (1940). Thickening of Calcium Carbonate Slurries. Industrial and Engineering Chemistry **32**, 663–667.

Cook, S. and Daniels, J. (1992). Essential Techniques for Object-Oriented Design. Proc. OOPS-59. London, UK: BCS OOPS Group.

Coplien, J. O. (1992). *Advanced C++*. Reading, Massachussets, USA: Addison-Wesley.

Corbin, J. R. (1991). *The Art of Distributed Applications*. New York, USA: Springer-Verlag.

Coutaz, J. (1987a). PAC, an Object-Oriented Model for Dialog Design. Proceedings of HCI Interact'87, North-Holland, Amsterdam, 431–436.

Coutaz, J. (1987b). The Construction of User Interfaces and the Object Paradigm. Proc. ECOOP'87, Paris, 121–130.

Coutaz, J. and Balbo, S. (1991). Applications: A Dimension Space for User Interface Management Systems. Proc. CHI'91 Conference, 27–32.

Cox, B. (1986). *Object-oriented Programming: an Evolutionary Approach*. Reading, Massachussets, USA: Addison-Wesley.

Crenshaw, J. W. (1991). A Perfect Marriage. Computer Language, June 1991, 44–54.

Crowther, C., Blevins, T. and Burns, D. (1987). A Lime Kiln Control Strategy to Maximize Efficiency and Energy Management. Appita **40**, 29–32.

Cugini, U. (1991). The Problem of User Interface in Geometric Modelling.

Computers in Industry **17**, 335-339.

Dahl, O. and Nygaard, K. (1966). Simula: An Algol-based Simulation Language. Communications of the ACM **9**, 671–678.

Davis, M. E. (1984). Numerical Methods and Modelling for Chemical Engineers. New York, USA: John Wiley and Sons.

Dean, H. (1991). Object-oriented Design Using Message Flow Decomposition. Journal of Object-Oriented Programming, May 1991, 21–31.

Decouchant, D., Krakowiak, S., Meysembourg, M., Riveill, M. and de Pina, X. R. (1989). A Synchronization Mechanism for Typed Objects in a Distributed System. SIGPLAN Notices **24**:**4**, 105–107.

Deininger, A. D. and Fernandez, C. V. (1990). Making Computer Behavior Consistent: the OSF/Motif Graphical User Interface. Hewlett-Packard Journal, June 1990, 6–12.

Dekkiche, E. A., Prévôt, P., En-Naamani, A. and Barbier, J. P. (1980). Modelling and Simulation of a Rotary Cement Kiln - Cybernetic Approach. Proc. IFAC 3, System Approach for Development, Rabat, Morocco, 101–118.

Delsanto, P. P., Kaniadakis, G. and Scalerandi, M. (1994). Time Scaling in the Parallel Processing Simulation of Diffusion Processes. Computers and Mathematics with Applications **27**, 51–61.

Dewan, P. and Solomon, M. (1990). An Approach to Support Automatic Generation of User Interfaces. ACM Transactions on Programming Languages and Systems **12**:4, 566-609.

Dodani, M. H., Hughes, C. E. and Moshell, J. M (1989). Separation of Powers. BYTE, March 1989, 255–262.

Dolado, J. J. (1991). Structured Development of Graph-Grammars for Icon Manipulation. ACM SIGSOFT Software Engineering Notes **16**, 46–51.

Dony, C. (1988). An Object-Oriented Exception Handling System for an Object-oriented Language. Proc. ECOOP'88, 146–161.

Dony, C. (1990). Exception Handling and Object-Oriented Programming: Towards a Synthesis. Proc. ECOOP/OOPSLA'90, 322–330.

Dony, C., Purchase, J. and Winder, R. (1992). Exception Handling in Object-Oriented Systems. OOPS Messenger **3**, 17–30.

Doraiswaimy, L. K. and Sharma, M. M. (1984). *Heterogeneous Reactions -*

Analysis, Examples, and Reactor Design, Vol.1 - Gas-Solid and solid-solid reactions, Chapters 19 and 20. New York, USA: A Wiley-Interscience Publication, John Wiley and Sons.

Dorris, G. M. and Allen, L. H. (1985). The Effect of Reburned Lime Structure on the Rates of Slaking, Causticizing and Lime Mud Settling. Journal of Pulp and Paper Science **11**:4, J89–J98.

Dorris, G. M. and Allen, L. H. (1986). Physicochemical Aspects of Recausticizing. 1986 Kraft Recovery Operations Seminar/Tappi Seminar Notes, 7–19.

Dorris, G. M. and Allen, L. H. (1987). Operating Variables Affecting the Causticizing of Green Liquors with Reburned Limes. Journal of Pulp and Paper Science **13**:3, J99–J105.

Dorris, G. M. (1990). Equilibrium of the Causticizing Reaction and its Effect on Slaker Control Strategies. 1990 Pulping Conference/TAPPI Proceedings, 245–251.

Dorris, G. M. (1993). The Physical Characterization of Hydrated Reburned Lime and Lime Mud Particles. Journal of Pulp and Paper Science **19**:6, J256–J267.

Dotson, B. E. and Krishnagopalan, A. (1990). Causticizing Reaction Kinetics. 1990 Pulping Conference/TAPPI Proceedings, 235–244.

Drury, H. A., Clark, K. W., Hermes, R. E., Feser, J. M., Thomas Jr., L. J. and Donis-Keller, H. (1992). A Graphical User Interface for Quantitative Imaging and Analysis of Electrophoretic Gels and Autoradiograms. BioTechniques **12**, 892-901.

Duce, D. A., Gomes, M. R., Hopgood, F. R. A. and Lee, J. R. (Eds.) (1991). *User Interface Management and Design*. Eurographics Seminar Series. Berlin: Springer-Verlag, 36–49.

Dubois-Pèlerin, Y., Zimmermann, T., and Bomme, P. (1992). Object-oriented Finite Element Programming: II. A Prototype Program in Smalltalk. Computer Methods in Applied Mechanics and Engineering **98**, 361–397.

Duimovich, J. and Milinkovich, M. (1991). Smalltalk and Embedded Systems. Dr. Dobb's Journal, October 1991, 86–95.

Dutta, S. and Shirai, T. (1980). Experimental Investigation on a Fast and Exothermic Solid-liquid Reaction System. Chemical Engineering Science **35**, 209–216.

Edwards, L. L., and Singh, P. (1984). Lime Kiln Simulation: Impact of Alternative Fuels and Dams. AIChE Symposium Series **80**, 74–86.

Ege, R. K. (1992). *Programming in an Object-oriented Environment.* San Diego, USA: Academic Press, Inc..

Elsila, M., Leiviska, K., Nettamo, K. and Pulkkinen, T. (1979). Computer Control of Causticizing and Lime Kiln Area is Possible. Pulp and Paper **53**, 152–159.

Encarnação, J., Cote-Muñoz, J., Eckardt, D., Rix, J. and Teixeira, J. (1991). User Interfaces to Support the Design Process. Computers in Industry **17**, 317–333.

Fikes, R. E. and Kehler, T. P. (1985). The Role of Frame-Based Representation in Reasoning. Communications of the ACM on Knowledge-Based Systems **28**, 904.

Filman, R. E. (1987). Retrofitting Objects. Proc. OOPSLA'87. Special Issue of SIGPLAN Notices **22**:12, 342–353.

Finlayson, B. (1980). *Non-linear Analysis in Chemical Engineering.* New York, USA: McGraw-Hill.

Fong, C. F. (1993). An Operational Approach to Object-oriented Software Development. PhD Dissertation, Aston University, UK.

Ford, W. and Chen, G. (1991). Overcoming Operation Problems in the Lime Kiln. 1991 Pulping Conference/ TAPPI Proceedings, 889–891.

Forde, B. W. R., Soschi, R. O. and Stiemer, S. F. (1990). Object-oriented Finite Element Analysis. Computers and Structures **34**, 355–374.

Foust, A. S., Wenzel, L. A., Clump, C. W., Maus, L.. and Andersen, L. B. (1980). *Principles of Unit Operations* (Second Edition). New York, USA: John Wiley and Sons.

Galtung, F. L. and Williams, T. J. (1969). A Survey of the Status of the Mathematical Modelling of the Chemical Recovery Section of a Kraft Paper Mill, Report Nr.23, Purdue Laboratory for Applied Industrial Control, Purdue University, Lafayette, Indiana, USA.

Gasser, L. (1989). MACE: High-Level Distributed Objects in a Flexible Testbed for Distributed AI Research. SIGPLAN Notices **24**:4, 108–110.

Gehani, N. and Roome, W. D. (1989). *The Concurrent C Programming Language.* Summit, New Jersey, USA: Silicon Press.

Geist, G. A., Sunderam, V. S. (1993). Network Based Concurrent Computing on the PVM System, Report of research performed at the Mathematical Sciences Section of Oak Ridge National Laboratory, Oak Ridge, Tenessee, USA.

Geist, G. A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. (1994). PVM 3 User's Guide and Reference Manual. Oak Ridge National Laboratory/TM-12187, Oak Ridge, Tenessee, USA.

Georgakis, C., Chang, C. W. and Szekely, Y. (1979). A Changing Grain Size Model for Gas-Solid Reactions. Chemical Engineering Science **34**, 1072–1075.

Gilmore, W. E., Gertman, D. I. and Blackman, H. S. (1989). *The User-Computer Interface in Process Control: a Human-Factors Engineering Handbook.* London, UK: Academic Press, Inc.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation.* Reading, Massachussets, USA: Addison–Wesley.

Goldberg, A. (1984). *Smalltalk-80: The Interactive Programming Environment.* Reading, Massachussets, USA: Addison–Wesley.

Gorlen, K., Orlow, S. M. and Plexico, P. S. (1990). *Data Abstraction and Object-oriented Programming in C++.* New York, USA: John Wiley and Sons Ltd.

Gorog, J. P. and Adams, T. N. (1987a). Design and Performance of Rotary Lime Kilns in the Pulp and Paper Industry: Part I – A Predictive Model for a Rotary Lime Reburning Kiln. 1987 Kraft Recovery Operations/Tappi Seminar Notes, 41–47.

Gorog, J. P. and Adams, T. N. (1987b). Design and Performance of Rotary Lime Kilns in the Pulp and Paper Industry: Part II – The Effect of Chain System Design and the Refractory Lining on Lime Kiln Performance. 1987 Kraft Recovery Operations/Tappi Seminar Notes, 48–54.

Gorog, J. P. and Adams, T. N. (1987c). Design and Performance of Rotary Lime Kilns in the Pulp and Paper Industry: Part III – How Flame Characteristics and Product Coolers Affect Lime Kiln Performance. 1987 Kraft Recovery Operations/Tappi Seminar Notes, 55–62.

Gorog, J. P. and Adams, T. N. (1987d). Design and Performance of Rotary Lime Kilns in the Pulp and Paper Industry: Part IV - The Effect of Operating Variables and O_2 Enrichment on Lime Kiln Performance. 1987 Kraft Recovery Operations/Tappi Seminar Notes, 63–68.

Gorog, J. P. and Adams, T. N. (1987e). Design and Performance of Rotary Lime Kilns in the Pulp and Paper Industry: Part V - Heat Transfer, Kiln

Geometry, and Lime Kiln Performance Rating. 1987 Kraft Recovery Operations/Tappi Seminar Notes, 69–73.

Graham, I. (1994). *Object-oriented Methods* (Second Edition). Wokingham, England: Addison–Wesley.

Gray, P. D. and Mohamed, R. (1990). *Smalltalk-80: A Practical Introduction.* London, UK: Pitman Publishing.

Green, M. (1985). The University of Alberta User Interface Management System. Computer Graphics **19**, 205–213.

Grogono, P. and Bennet, A. (1989). Polymorphism and Type Checking in Object-oriented Languages. SIGPLAN Notices **24**:11, 109–115.

Grudin, J. (1992). Utility and Usability: Research Issues and Development Contexts. Interacting with Computers **4**, 209–217.

Guimarães, J. C. (1980). PhD Dissertation, University of Idaho, USA.

Guimarães, J. C. and Edwards, L. L. (1981). Lime Kiln Energy Savings: Modelling, Simulation and Control. AIChE Symposium Series **77**, 122–127.

Guimarães, J. C., Lima, A. F., Park, S. W. and Yojo, L. M. (1986). Uso da Simulação no Controle do Forno da Cal por Microcomputador. O Papel, Fevereiro 1986, 29–36. (In Portuguese)

Harrison, M. and Thimbleby, H. (1990). The Role of Formal Methods in Human-Computer Interaction. In *Formal Methods in Human-Computer Interaction.* M. Harrison and H. Thimbleby (Eds.), Cambridge, UK: Cambridge University Press.

Harrison, M. and Dix, A. (1990). A State Model of Direct Manipulation in Interactive Systems. In *Formal Methods in Human-Computer Interaction.* M. Harrison and H. Thimbleby (Eds.), Cambridge, UK: Cambridge University Press.

Hasset, N. J. (1964–1965). Mechanism of Thickening and Thickener Design. Transactions of the Institution of Mining and Mettalurgy. **74**, 627–656.

Hayes, F. and Baran, N. (1989). A Guide to GUI's. BYTE, July 1989, 250–257.

Heller, D. (1993). *XView Programming Manual for version 3.2*, Vol.7 (Third Edition, updated for XView version 3.2). Sebastopol, California, USA: O'Reilly and Associates.

Helm, R., Holland, I. M. and Gangopadhyay, D. (1990). Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. Proc. ECOOP/ OOPSLA'90, 169–180.

Henderson-Sellers, B. and Edwards, J. M. (1990). The Object-oriented Systems Life Cycle. Communications of the ACM **33**:9, 142–159.

Hindel, B. (1989). An Object-oriented Programming Language for Distributed Systems: HERAKLIT. SIGPLAN Notices **24**:4, 114–116.

Hindmarsh, A. C. (1980). Lsode and Lsodi: Two New Initial Value Ordinary Differential Equation Solvers. ACM Signum Newsletters **15**, 10–11.

Hix, D. and Hartson, H. R. (1993). *Developing User Interfaces: Ensuring Usability through Product and Process.* New York, USA: John Wiley and Sons.

Hix, D. and Schulman, R. S. (1991). Human-computer Interface Development Tools: a Methodology for Their Evaluation. Communications of the ACM **34**:3, 75-87.

Hoeber, T. (1988). Face to Face with OpenLook. BYTE, December 1988, 286–296.

Hsieh, J. S., Davey, K. R., Vatchtsevanos and Cheng, C. H. (1990). Mud filter Down Time on Lime Kiln Energy Efficiency. 1990 Pulping Conference/TAPPI Proceedings, 253–258.

Hypponen, O. and Luukko, A. (1984). The Residence Time Distributions of Liquor and Lime Flows in the Recausticizing Process. Tappi Journal **67**:7, 46–48.

Ince, D. (1991). *Object-Oriented Software Engineering with C++.* London, UK: McGraw-Hill Book Company.

Ishida, M. and Wen, C. Y. (1968). Comparison of Kinetic and Diffusional Models for Gas-Solid Reactions. AIChE Journal **14**, 311–317.

Jacob, R. J. K. (1983). Using Formal Specifications in the Design of a Human-Computer Interface. Communications of the ACM **26**:4, 259–264.

Jacob, R. J. K. (1986). A Specification Language for Direct-Manipulation User Interfaces. ACM Transactions on Graphics **5**:4, 283–317.

Jacobi, E. F. and Williams, T. J. (1973a). A Dynamic Model and Advanced Direct Digital Control System for a Kraft Mill Liquor Preparation System. Report n.56, Vol.I - Narrative, Purdue Laboratory for Applied Industrial Control, Purdue University, USA.

Jacobi, E. F. and Williams, T. J. (1973b). A Dynamic Model and Advanced Direct Digital Control System for a Kraft Mill Liquor Preparation System. Report n.56, Vol.II - Appendices, Purdue Laboratory for Applied Industrial Control, Purdue University, USA.

Jacobson, B. (1992). The Ultimate User Interface. BYTE, April 1992, 175–182.

James, S. (1992). Lime Kiln Operations. 1992 Kraft Recovery Operations Short Course/ TAPPI Notes, 73–89.

Jazayeri, M. (1989). Objects for Distributed Systems. SIGPLAN Notices **24**:4, 117–119.

Jeffries, R. and Desurvire, H. (1992). Usability Testing vs. Heuristic Evaluation: Was There a Contest? SIGCHI Bulletin **24**, 39–52.

Jensen, L. and King, A. J. (1992). Frontier: A Graphical Interface for Portfolio Optimization in a Piecewise Linear-quadratic Risk Framework. IBM Systems Journal **31**, 62-70.

Jordan, D. (1990). Implementation Benefits of C++ Language Mechanisms. Communications of the ACM **33**:9, 61–64.

Judge, P. (1990). Just a Regular GUI. Systems International, March 1990, 57–58.

Kaiser, G. E. (1989). Concurrent Meld. SIGPLAN Notices **24**:4, 120–122.

Kamath, Y. H., Smilan, R. E. and Smith, J. G. (1993). Reaping Benefits with Object-Oriented Technology. AT&T Technical Journal, September/October 1993, 14–24.

Kantorowitz, E. and Sudarsky, O. (1989). The Adaptable User Interface. Communications of the ACM **32**:11, 1352-1358.

Karaorman, M. and Bruno, J. (1993). Introducing Concurrency to a Sequential Language. Communications of the ACM **36**:9, 103–116.

Karplus, W. J. (1977). The Spectrum of Mathematical Modelling and Systems Simulation. Mathematics and Computers in Simulation **19**, 3-10.

Keene, S. E. (1989). *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS.* Ch.12. Reading, Massachussets, USA: Addison-Wesley.

Kempf, R., Stelzner, M. (1987). Teaching Object-oriented Programming with the KEE System. Proc. OOPSLA'87. Special Issue of SIGPLAN Notices **22**:12, 11–25.

Kernighan, B. W. and Ritchie, D. M. (1978). *The C Programming Language.* Englewood Cliffs, New York: Prentice Hall.

Kimbrell, R. E., Correll, L. and Bass, R. (1989). The QSIM Simulation Toolkit. BYTE, July 1989, 259–266.

King, G. B. and Horlick, G. (1992). An Advanced User Interface for Processing and Displaying Interferometric and Spectral Data: SpectroPlot. Spectrochimica Acta **47B**, E353-E370.

Kinzner, K. (1968). Investigations in the Causticizing of Green Liquors. Proc. Symposium on the Recovery of Pulping Chemicals, International Union of Pure and Applied Chemistry, Pulp, Paper and Board Section, Helsinki, Finland, 279–302 (In German).

Knolle, N. (1989). Why Object-oriented User Interface Toolkits are Better. Journal of Object-oriented Programming, November/December 1989, 63–67.

Koegel, J. F. (1989). Parallel Objects on Distributed Constraint Logic Programming Machines. SIGPLAN Notices **24**:4, 123–125.

Koenig, A. and Stroustrup, B. (1990). Exception Handling for C++. USENIX C++ Conference Proceedings, April 1990, 149–176.

Koivo, A. J. and Chase, P. M. (1972). Determination of a Mathematical Model for a Lime Kiln. Proc. 3rd IFAC/IFIP Conference on Digital Computer Application to Process Control, Papper XII-3, Helsinki, 1–6.

Kojo, M. (1979a). Causticizing Reaction Equilibrium and Mill Operating Practices. Paperi ja Puu — Papper och Trä **61**, 244–250.

Kojo, M. (1979b). Some Theoretical and Practical Aspects of the Causticizing Equilibrium and Raising the White Liquor Concentration. Paperi ja Puu — Papper och Trä **61**, 443–447.

Kojo, M. (1979c). Effects of the Changing White Liquor Concentration on the Operation, Power Consumption and Investment of the Recovery Process. Paperi ja Puu — Papper och Trä **61**, 701–709.

Kojo, M. (1980). Benefits can be Obtained with a Stronger and Hotter White Liquor. Pulp and Paper **54**, 174–176.

Kramm, D. J. (1979). Update on Lime Sludge Kilns in the Pulp Mill Environment. Paper Trade Journal **163**, 23–27.

Kramm, D. J. and Schultz, W. L. (1985). In *Chemical Recovery in the Alkaline Pulping Process*, Ch.6, G. Hough (Ed.). USA: Tappi Press.

Krasner, G. E. and Pope, S. T. (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-oriented Programming, August/September 1988, 26–49.

Kreutzer, W. (1986). *System Simulation Programming Styles and Languages.* Sydney, Australia: Addison-Wesley Publishing Company.

Kuo, F.-Y. and Karimi, J. (1988). User Interface Design from a Real-Time Perspective. Communications of the ACM **31**:12, 1456–1466.

Kynch, G. J. (1952). A Theory of Sedimentation. Transactions of the Faraday Society **48**, 166– 176.

Laffra, C. and van den Bos, J. (1990). A Layered Object-Oriented Model for Interaction. In *Advances in Object-Oriented Graphics*, Ch.4, Eurographics Seminar Series. Springer-Verlag, 47–65.

Lainhart, T. (1991). Intrinsics of the X Toolkit. Dr. Dobb's Journal, February 1991, 94–129.

Lantz, K. A., Tanner, P. P., Binding, C., Huang, K.-T. and Dwelly, A. (1987). Reference Models, Window Systems, and Concurrency. Computer Graphics **21**, 87–97.

Lapalme, G. (1989). Plasma-II: an Actor Approach to Concurrent Programming. SIGPLAN Notices **24**:4, 81–86.

Lazarev, G. L. (1991). Reusability in Smalltalk: A Case Study. Journal of Object-oriented Programming, May 1991, 11–20.

Lea, R., Jacquemot, C. and Pillevesse, E. (1993). COOL: System Support for Distributed Programming. Communications of the ACM **36**:9, 37–46.

Leiviska, K., Matti, S. and Uronen, P. (1987). Causticizing Reaction and Methods to Analyse its Proceeding, Report 105, Department of Process Engineering, University of Oulu, Oulu, Finland.

Levenspiel, O. (1972). *Chemical Reaction Engineering* (Second Edition). New York, USA: John Wiley and Sons.

Levenspiel, O. (1993). *The Chemical Reactor Omnibook.* Corvallis, Oregon, USA: OSU Book Stores, Inc.

Lewis, D. R., (1979). Fabric Filters for Kraft Mill Lime Kilns. Pulp and Paper Canada **80**, 67–74.

Lieberman, H. (1985). There's More to Menu Systems than Meets the Screen. Computer Graphics **19**, 181–189.

Lima, AF., Park, S. W. and Yojo, L. M. (1983). Variáveis Operacionais que Afetam o Consumo de Combustíveis no Forno de Cal - Parte I. Proceedings of the III Congresso Latino-Americano de Celulose e Papel, São Paulo, Brasil, 865–876. (In Portuguese)

Lindberg, H. and Ulmgren, P. (1983). Equilibrium Studies of White Liquor Preparation in Kraft Mills. Journal of Pulp and Paper Science **9**:3, TR7–TR12.

Lindberg, H. and Ulmgren, P. (1986). The Chemistry of the Causticizing Reaction - Effects on the Operation of the Causticizing Department in a Kraft Mill. Tappi Journal **69**:3, 126–130.

Linton, M. A., Vlissides, J. M. and Calder, P. R. (1989). Composing User Interfaces with InterViews. Computer **22**, 8–22.

Löhr, K.-P. (1993). Concurrency Annotations for Reusable Software. Communications of the ACM **36**:9, 81–89.

Lyons, J. W., Min, H. S., Parisot, P. E. and Paul, J. F. (1962). Industrial and Engineering Chemistry Process Design and Development **1**, 29 –33.

McIlwain, J. A. (1992). Kiln Control. Pulp and Paper Canada **93**:11, T307–T310.

Manning, C. (1989). A Peek at Acore, an Actor Core Language. Sigplan Notices **24**:4, 84–86.

Marcus, A. and Van Dam, A. (1991). User Interface Developments for the Nineties. Computer, September 1991, 49–57.

Marquardt, W. (1992). An Object-oriented Representation of Structured Process Models. Computers and Chemical Engineering **16**, 329–336.

Masui, T. (1991). User Interface Construction Based on Parallel and Sequential Execution Specification. IEICE Transactions **E74**, 3168–3179.

Mehra, N. K. (1981). Use of Two-Stage Lime Mud Washing Can Control Kiln TRS Emissions. Pulp and Paper **55**, 134–137.

Mehra, N. K., Cornell, C. F. and Hough, G. W. (1985). in *Chemical Recov-*

ery in the Alkaline Pulping Process, Ch.5., G. Hough (Ed.). USA: Tappi Press.

Menétrey, P. H. and Zimmermann, T. H. (1993). Object-oriented Non-Linear Finite Element Analysis: Application to J2 Plasticity. Computers and Structures **49**, 767–777.

Menga, G., Picchiottino, P., Gallo, P. and Lo Russo, G. (1991). Framework for Object-oriented Design and Prototyping of Manufacturing Systems. Journal of Object-oriented Programming, Focus on Analysis and Design, 41–53.

Meyer, B. (1988). *Object-oriented Software Construction*. London: Prentice-Hall.

Meyer, B. (1989). From Structured Programming to Object-oriented Design: The Road to Eiffel. Structured Programming **10**, 19–39.

Meyer, B. (1993). Systematic Concurrent Object-oriented Programming. Communications of the ACM **36**:9, 56–80.

Micallef, J. (1988). Encapsulation, Reusability and Extensibility in Object-oriented Programming Languages. Journal of Object-oriented Programming, April/May 1988,12–35.

Michaels, A. S. and Bolger, J. C. (1962). Settling Rates and Sediment Volumes of Flocculated Kaolin Suspensions. Industrial and Engineering Chemistry Fundamentals **1**, 24–33.

Miller, J. D. (1990). *An Open Look at UNIX*. Hemel Hempstead, UK: Prentice-Hall International (UK) Limited.

Mitchell, R. J. (1993). *C++ Object-oriented Programming*. Basingstoke, Hampshire, UK: The MacMillan Press Ltd.

Mitrani, I. (1982). *Simulation Techniques for Discrete Event Systems*. Cambridge Computer Science Texts –14. Cambridge, UK: Cambridge University Press.

Mittet, G. R. (1994). Factors Affecting Slaker Control in Causticizing Plants. 1994 Process Control Symposium/TAPPI Proceedings, 155–165.

Montazemi, A. R. (1991). The Impact of Experience on the Design of User Interface. International Journal of Man-Machine Studies **34**, 731–749.

Moore, D. (1990). The Migration of the X Window System. BYTE IBM Special Edition, Fall 1990, 183-185.

Mühlhäuser, M., Gerteis, W. and Heuser, L. (1993). DOCASE: A Methodic

Approach to Distributed Programming. Communications of the ACM **36**:9, 127–138.

Mullin, M. (1990). *Object-Oriented Program Design with Examples in C++.* Third Printing. Reading, Massachussets, USA: Addison-Wesley Publishing Company, Inc.

Mumford,W. G., Smith, D. B., Edwards, L. L. and Vegega, A. M. (1989). Reducing Lime Kiln Fuel Usage: Retrofit Results from Actual Mill Installations. Tappi Journal **72**:1, 29–33.

Myers, B. A. and Rosson, M. B. (1991). User Interface Programming Survey. Sigplan Notices **26**:8, 19–22.

Navon, I. M. and Cai, Y. (1993). Domain Decomposition and Parallel Processing of a Finite Element Model of the Shallow Water Equations. Computer Methods in Applied Mechanics and Engineering **106**, 179–212.

Nicholls, B. (1990). Looking at the Graphical User Interface. BYTE IBM Special Edition, Fall 1990, 161–166.

Nicholson, R. T. (1991). Designing a Portable GUI Toolkit. Dr. Dobb's Journal, January 1991, 68–75.

Nielsen, J. (1993). *Usability Engineering.* London, UK: Academic Press, Inc.

Nielson, G. M. (1991). Visualization in Scientific and Engineering Computation. Computer, September 1991, 58–66.

Notidis, E. and Tran, H. (1993). Survey of Lime Kiln Operation and Ringing Problems. Tappi Journal **76**:5, 125–131.

Nye, A. (1990). *Xlib Programming Manual for version X11.* Vol.I (Sixth Printing). Sebastopol, California, USA: O'Reilly and Associates.

Nye, A. (Editor) (1990a). Xlib Reference Manual for version X11. Vol.II, Sixth Printing. Sebastopol, California, USA: O'Reilly and Associates.

ObjectVision (1990). ObjectVision Package and Manual, ObjecVision Inc.

Olsen, J. C. and Direnga, O. G. (1941). Settling Rate of Calcium Carbonate in the Causticizing of Soda Ash. Industrial and Engineering Chemistry **33**, 204–218.

Orr, C. (1977). *Filtration — Principles and Practices, Part I.* Chemical Processing and Engineering, Vol.10. L. F. Albright, R. N. Maddox and J. J. Macketta (Eds.). New York, USA: Marcel Dekker, Inc.

Pais, F. I. C. C. and Portugal, A. A. T. G. (1993a). A Mathematical Model for the Lime Kiln. Proc. CHEMPOR'93, Porto, Portugal, 669–681.

Pais, F. I. C. C. and Portugal, A. A. T. G. (1993b). The Lime Kiln: a Dynamic Study. Proc. ICDERS, International Colloquium on the Dynamics of Explosive and Reactive Systems, Coimbra, Portugal, C2.4.1–C2.4.13.

Pais, F. I. C. C. and Portugal, A. A. T. G. (1994a). Determination of the Steady-State of Isothermal Two-Phase Continuous Stirred Tank Reactors. Chemical Engineering Science **49**, 3447–3456.

Pais, F. I. C. C. and Portugal A. A. T. G. (1994b). A Mathematical Model for Non-catalytic Liquid-solid Reversible Reactions. Computers and Chemical Engineering (Submitted).

Pais, F. I. C. C. and Portugal, A. A. T. G. (1994c). A Dynamic Model for the Causticizing Units. Journal of Pulp and Paper Science (Submitted).

Pais, F. I. C. C., Gay, B. and Portugal, A. A. T. G. (1994). XProcSim: An X Window-based GUI for the Chemical Recovery Cycle of a Paper Pulp Mill. In *Human-Machine Communication for Educational Systems Design.* M. Brouwer-Janse and T. Harrington (Eds.). Berlin: Springer-Verlag.

Pangalos, G. J. (1992a). Standardization of the User Interface. Computer Standards and Interfaces **14**, 223–229.

Pangalos, G. J. (1992b). Consistency and Standardization of User Interfaces. Information and Software Technology **34**, 397-401.

Papathomas, M. (1989). Working Group on Models for Concurrent Object-based Programming. Sigplan Notices **24**:4, 9–11.

Park, J. Y. and Levenspiel, O. (1975). The Crackling Core Model for the Reaction of Solid Particles. Chemical Engineering Science **30**, 1207–1214.

Park, J. Y. and Levenspiel, O. (1977). The Crackling Core Model for the Multistep Reaction of Solid Particles. Chemical Engineering Science **32**, 233–234.

Pearce, K. W. (1973). A Heat Transfer Model for Rotary Kilns. Journal of the Institute of Fuel, December 1973, 363–371.

Peddie, J. (1992). *Graphical User Interfaces and Graphics Standards.* New York, USA: McGraw-Hill Inc.

Perlman, G. (1988). Teaching User Interface Development to Software Engineers. Proc. of the Human Factors Society - 32nd Annual Meeting, 391–394.

Perry, R. E. (1989). Lime Kiln Energy Conservation. Tappi Journal **72**:6, 243–246.

Perry, R. H. and Chilton, C. H. (Eds.) (1973). *Chemical Engineers' Handbook* (Fifth Edition). McGraw-Hill Book Company.

Peskin, R., Walther, S. S. and Froncioni, A. M. (1989). Smalltalk — The Next Generation Scientific Computing Interface? Mathematics and Computers in Simulation **31**, 371–381.

Petty, C. A. (1975). Continuous Sedimentation of a Suspension with a Nonconvex Flux Law. Chemical Engineering Science **30**, 1451–1458.

Pokkunuri, B. P. (1989). Object-oriented Programming. SIGPLAN Notices **24**:11, 96–101.

Portugal, A. A. T. G. and Pais, F. I. C. C. (1989). A Mathematical Model for the Causticizing Reaction. Proc. CHEMPOR'89, Lisbon, 8R1–8R8.

Pountain, D. (1989). The X Window System. BYTE, January 1989, 353–360.

Purchase, J. A. and Winder, R. L. (1991). Debugging Tools for Object-Oriented Programming. Journal of Object-Oriented Programming, June 1991,10–27.

Ragnemalm, E. (1994). Intelligent Assistance for Simulator-Based Training. In *Human-Machine Communication for Educational Systems Design.* M. Brouwer-Janse and T. Harrington (Eds.), Berlin: Springer-Verlag.

Ramachandran, P. A. and Doraiswamy, L. K. (1982). Modelling of Noncatalytic Gas - Solid Reactions. AIChE Journal **28**, 881-900.

Ranade, P. V. and Harrison, D. P. (1979). The Grain Model Applied to Porous Solids with Varying Structural Properties. Chemical Engineering Science **34**, 427–432.

Reiss, L. and Radin, J. (1992). *X Window Inside and Out.* Berkeley, California, USA: Osborne McGraw-Hill.

Ribeiro, B., Henriques, J. and Dourado, A. (1992). Simulação Computacional dos Evaporadores e Forno de Cal: Apoio ao Treino de Operação e ao Controlo Automático. Proc. Tecnicelpa, 185.

Richardson, J. F. and Zaki, W. N. (1954). Sedimentation and Fluidization.

Transactions of the Institution of Chemical Engineers **32**, 35–53.

Richardson, B., Watkinson, A. P. and Barr, P. V. (1990). Combustion of Lignin in a Pilot Lime Kiln. 1990 Pulping Conference/ TAPPI Proceedings, 457–465.

Rimmer, S. (1992). *Graphical User Interface Programming.* Blue Ridge Summit, Pensylvannia, USA: Windcrest-McGraw-Hill.

Rubin, T. (1988). *User Interface Design for Computer Systems.* Chichester, UK: Ellis Horwood Limited.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991). *Object-Oriented Modelling and Design.* Englewood Cliffs, New Jersey, USA: Prentice-Hall.

Rushton, A. and Katsoulas, C. (1984). *Practical and Theoretical Aspects of Constant Pressure and Constant Rate Filtration.* In *Solid-Liquid Separation*, J. Gregory (Ed.), Chichester, UK: Ellis Horwood Limited.

Rydin, S., Haglund, P. and Mattson, E. (1977). Causticizing of Technical Green Liquors with Various Lime Qualities. Svensk Papperstidning, Nr.2, 54–58.

Rydin, S. (1978). The Kinetics of the Causticizing Reaction. Svensk Papperstidning Nr.2, 43–48.

Sass, A. (1967). Simulation of the Heat-Transfer Phenomena in a Rotary Kiln. Industrial and Engineering Chemistry Process Design and Development **6**, 532–535.

Satterfield, C. N. and Feakes, F. (1959). Kinetics of the Thermal Decomposition of Calcium Carbonate. AIChE Journal **5**, 115–122.

Scheifler, R. W. and Gettys, J. (1986). The X Window System. ACM Transactions on Graphics **5**:2, 79–109.

Schiele, F. and Green, T. (1990). HCI Formalisms and Cognitive Psychology: the Case of the Task Action Grammar. In *Formal Methods in Human-Computer Interaction.* M. Harrison and H. Thimbleby (Eds.). Cambridge, UK: Cambridge University Press.

Schmucker, K.J. (1986). MacApp: an Application Framework. BYTE **11**, 189–193.

Schmucker, K. 1988) Using Objects to Package User Interface Functionality. Journal of Object-oriented Programming, April/May 1988, 40–45.

Scott, K. J. (1968). Thickening of Calcium Carbonate Slurries. Industrial and Engineering Chemistry Fundamentals **7**, 484–490.

Serpent Overview (1989). Software Engineering Institute, Carnegie Mellon University, CMU/SEI-89-UG-2, Pittsburgh, Pennsylvannia, USA.

Shan, Y.-P. (1990a). MoDE: A UIMS for Smalltalk. Proc. ECOOP/ OOP-SLA'90, 258–268.

Shan, Y.-P. (1990b). An Object-Oriented Framework for Direct-Manipulation User Interfaces. In *Advances in Object-Oriented Graphics*, Eurographics Seminar Series, Springer-Verlag, 3–19.

Shandle, J. (1990). Real-time Simulation Speeds HDTV Designs. Electronics, March 1990, 68-69.

Shannon, P. T., Stroupe, E. and Tory, E. (1963). Batch and Continuous Thickening. Industrial and Engineering Chemistry Fundamentals **2**, 203–211.

Shaw, J. F. G. (1992). The Integration of Process Simulation and Engineering Design. Computers and Chemical Engineering **16**, 465–472.

Shewchuk, C. F., Leaver, E. W. and Schindler, H. E. (1991). The Roles for Process System Simulation in the Design, Testing and Operation of Processes, Process Control and Advanced Control Systems. Pulp and Paper Canada **92**:9, 53–59.

Shneiderman, B. (1983). Direct Manipulation: a Step Beyond Programming Languages. Computer **16**, 57–69.

Shneiderman, B. (1992). *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Reading, Massachussets, USA: Addison-Wesley.

Sibert, J. L., Hurley, W. D. and Bleser, T. W. (1986). An Object-Oriented User Interface Management System. Computer Graphics **20**, 259–268.

Smith, D. C., Irby, C., Kimball, R., Verplanck, B. and Harslem, E. (1982). Designing the Star User Interface. Byte Magazine **7**:4, 652–661.

Smith, S. L. and Mosier, J. N. (1986). Guidelines for Designing User Interface Software. The MITRE Corporation, Bedford, Massachssets, USA. (Public Release; Distribution Unlimited).

Smmok, G. A. (1986). *Handbook for Pulp and Paper Technologists.* M. J. Kocurek (Ed.): TAPPI/CPPA.

Snyder, A. (1986). Encapsulation and Inheritance in Object-Oriented Programming Languages. Proc. OOPSLA'86, 38–45.

Sommerville, I. (1992). *Software Engineering.* Reading, Massachussets, USA: Addison-Wesley.

Spang, H. A. (1972). A Dynamic Model of a Cement Kiln. Automatica **8**, 309–323.

Stamatakis, K. and Tien, C. (1991). Cake Formation and Growth in Cake Filtration. Chemical Engineering Science **46**, 1917–1933.

Steel, D. (1992a). A Remote Invocation Mechanism for Smalltalk and C++. Technical Report No.578, Department of Computer Science, University of London, 1–36.

Steel, D. (1992b). Operating Systems Support for Distributed Object Oriented Programming. Technical Report No.594, Department of Computer Science, University of London, 1–20.

Stephanopoulos, G., Jonhston, J., Kriticos, T., Lakshmanan, R., Mavrovouniotis, M. and Siletti, C. (1987). Design-Kit: An Object-oriented Environment for Process Engineering. Computers and Chemical Engineering **11**, 655–674.

Stiles, D. V. (1991). Recent Refractory Trends in Lime Recovery Kilns: Factors Affecting These Trends. 1991 Kraft Recovery Operations/TAPPI Short Course, 61–65.

Stroustrup, B. (1986). *The C++ Programming Language.* Reading, Massachussets, USA: Addison-Wesley.

Sutcliffe, A. G. and McDermott, M. (1991). Integrating Methods of Human-Computer Interface Design with Structured Systems Development. International Journal of Man-Machine Studies **34**, 631-655.

Sutherland, I. E.(1963). Sketchpad: A Man-Machine Graphical Communication System. Proc. AFIPS Spring Joint Computer Conference, 329–346.

Szekely, P. (1990). Template-Based Mapping of Application Data to Interactive Displays. Proc. UIST'90, 1–9.

Symbolics Inc., *Symbolics CommonLISP Language Dictionary.* Cambridge, Massachussets: Symbolical Inc.

Theliander, H. and Grén, U. (1987). A System Analysis of the Chemical Recovery Plant of the Sulphate Pulping Process, Part IV - Comments on the Influence of the Separation Properties in Slaking and Causticizing. Nordic Pulp and Paper Research Journal **2**:3, 109–115.

Thimbleby, H. (1990). *User Interface Design.* Wokingham, England: Addison-Wesley.

Think C^{TM}, The Professional's Choice (1991). Object-oriented Programming Manual, Symantec Corporation.

Thomas, D. A., LaLonde, W. R., Duimovich, J. and Wilson, M. (1989). Actra - A Multitasking/Multiprocessing Smalltalk. SIGPLAN Notices, **24**:4, 87–90.

Tomlinson, C., Kim, W., Scheevel, M., Singh, V., Will, B. and Agha, G. (1989). Rosette: An Object-oriented Concurrent Systems Architecture. SIGPLAN Notices **24**:4, 91–94.

Tory, E. M. (1961). PhD Dissertation, Purdue University, Indiana, USA.

Tracy, K. D. and Keinath, T. M. (1973). Dynamic Model for Thickening of Activated Sludge. AIChE Symposium Series **70**, 291–308.

Tran, H. N. and Barnham, D. (1990). Ringing Problems in Lime Kilns. Kraft Recovery Operations/Tappi Short Course Notes, 75–78.

Tran, H. N., Barham, D. and Reeve, D. W. (1990). Chloride and Potassium in the Kraft Chemical Recovery Cycle. Pulp and Paper Canada **91**:5, T185–T190.

Tran, H., Griffiths, J. and Budge, M. (1991). Experience of Lime Kiln Ringing Problems at E.B. Eddy Forest Products. Pulp and Paper Canada **92**:1, 78–82.

Treu, S. (1988). Designing a "Cognizant Interface" Between the User and the Simulation Software. Simulation, December 1988, 227-234.

Tsai, G.-J. and Tsao, G. T. (1991). Dynamic Response and Parameter Estimation in a Two-phase Continuous Flow Stirred Tank Reactor. Chemical Engineering Science **46**, 881–888.

Tufte, E. (1992). The User Interface: the Point of Competition. Bulletin of the American Society for Information Science, June/July 1992, 15–17.

Turc, M., Thomae, V. and Moisa, F. (1982). Analiza Procesului de Caustificare pe baza Modelului Matematic. Celuloza si Hirtie, Anul 31 Nr.1, 22-27 (In Romanian).

Udell, J. (1990). Smalltalk-80 Enters the Nineties. BYTE, October 1990, 138–142.

Ungar, D. and Smith, R. R. (1987). Self: The Power of Simplicity. Proc. OOPSLA'87. Special Issue of SIGPLAN Notices **22**:12, 227–243.

Urlocker, Z. (1989). Abstracting the User Interface. Journal of Object-oriented Programming, November/December 1989, 68–74.

Uronen, P. and Aurasmaa, H. (1979). Modelling and Simulation of Causticizing Plant and Lime Kiln. Pulp and Paper Canada **80**, 90–93.

Uronen, P., Aurasmaa, H. and Leiviska, K. (1976). Static and Dynamic Modelling of a Lime Circulation Loop. Paperi ja Puu **58**, 1–5.

Uronen, P. (1985). Modelling and Advanced Control of Lime Mud Filters. 1985 Pulping Conference/TAPPI Proceedings, 611–617.

Van der Meulen, P. S. (1987). INSIST: Interactive Simulation in Smalltalk. Proc. OOPSLA'87. Special Issue of SIGPLAN Notices **22**:12, 11-25.

Vegeais, J. A. and Stadtherr, M. A. (1992). Parallel Processing Strategies for Chemical Process Flowsheeting. AIChE Journal **38**, 1399–1407.

Villadsen, J. and Michelsen, M. L. (1978). *Solution of Differential Equation Models by Polynomial Approximation.* Englewood Cliffs, New York, USA: Prentice Hall Inc.

Wang, H. and Green, M. (1991). An Event-Object Recovery Model for Object-oriented User Interfaces. Proc. UIST'91, Hilton Head, South Carolina, USA, 107–115.

Wasserman, A. I. and Pircher, P. A. (1991). From Object-oriented Analysis to Design. Journal of Object-oriented Programming, September 1991, 46–50.

Wegner, P. (1987). Dimensions of Object-based Language Design. Special Issue of SIGPLAN Notices **22**:12, 342–353.

Weinand, A., Gamma, E. and Marty, R. (1988). ET++ – An Object-oriented Application Framework in C++. Proc. OOPSLA'88, 46–57.

Welty, J. R., Wicks, C. E. and Wilson, R. E. (1984). *Fundamentals of Momentum, Heat and Mass Transfer.* Chichester: John Wiley and Sons, Inc.

Williams, M., Smith, S. and Pecelli, G., (1990). Computer-Human Interface Issues in the Design of an Intelligent Workstation for Scientific Visualization. SIGCHI Bulletin, April 1990, 44–49.

Williams, G. (1983). The Lisa Computer System. Byte, February 1983, 33–50.

Wilson, S. (1990). The Object-Oriented Approach. IEE Review, July/August 1990, 277–280.

Wirfs-Brock, A. (1991). Getting to the Mainstream. Journal of Object-oriented Programming, May 1991, 51–54.

Wright, J. R., Benabdallah, S. and Engel, B. A. (1990). A Normalized User Interface for Complex Models. AI Applications **4**, 11–17.

Xu, W. and Dollimore, J. (1992). Transparent Remote Invocation for Distributed Object-Oriented Programming. Department of Computer Science, University of London, Technical Report No.603, 1–28.

Yagi, S. and Kunii, D. (1955). Studies on Combustion of Carbon Particles in Flames and Fluidized Beds. 5^{th} Symposium (International) on Combustion. Reinhold, New York, 231- 244.

Yip , C. W. and Dessey, R. E. (1991). Object Oriented Programing (OOP) and Graphical User Interfaces (GUI). Chemometrics and Intelligent Laboratory Systems **11**, 251–254.

Yoon, D. H. H. (1993). The Categorical Framework of Object-oriented Concurrent Systems. Computers and Mathematics with Applications **25**:2, 33–38.

Zeigler, B. P. (1976). *Theory of Modelling and Simulation.* New York, USA: John Wiley and Sons.

Zimmermann, T., Dubois-Pèlerin, Y. and Bomme, P. (1992). Object-oriented Finite Element Programming: I. Governing Principles. Computer Methods in Applied Mechanics and Engineering **98**, 291–303.

Zobel, R. N. and Lee, K. H. (1992). Graphical User Interface Based Simulation System for Mixed Application Areas. Proc. 24th Annual Summer Computer Simulation Conference, July 1992, 62-66.

Appendix A

The Lime Kiln

A.1 Introduction

The lime kiln has been used since the 1910's, and first made its appearance in the paper industry in North America in the mid 1920's (Kramm, 1979). It is usually a rotary type unit where drying and calcination of the lime mud from the causticizing plant take place. Calcination is described by the following equation

$$CaCO_3(s) \rightarrow CaO(s) + CO_2(g) + (-\Delta H_C) \tag{A.1}$$

yielding calcium oxide which is the main reactant in the transformation of green liquor into white liquor. The other main phenomenon which takes place is the evaporation of water from the lime mud, according to

$$H_2O(l) \rightarrow H_2O(g) + (-\Delta H_W) \tag{A.2}$$

Fuel must be burnt in order to supply the necessary energy for the endothermic decomposition reaction to take place.

The lime mud (a mixture of lime, basically calcium carbonate, and water) enters the kiln in what will henceforth be referred to as the solids feed end. The gaseous stream (a mixture of fuel and air) enters the kiln in the other end so that the two streams flow in counter-current. A chain system at the solids feed end assists in the evaporation of water from the lime feed, and also acts as a curtain or dust arrestor. A schematic representation of the lime kiln is presented in Fig.A.1. Detailed descriptions of current kiln design are common in

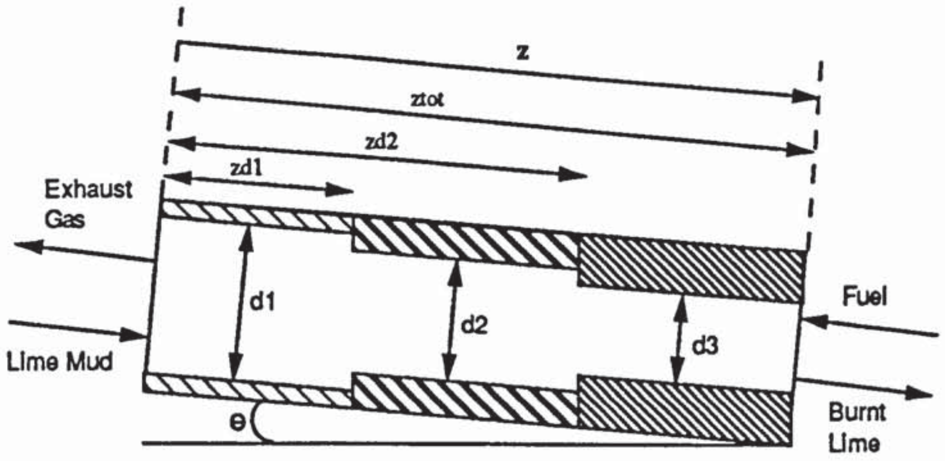


Figure A.1: Schematic representation of the lime kiln.

the literature (Kramm and Schultz, 1985), as well as descriptions of existing industrial installations (James, 1992). A privileged trend of current research is the possibility of using alternative fuels to reduce the dependency of the mill from external purchase of expensive fossil fuel (e.g. Richardson *et al.*, 1990). Another major subject of research is the study of new refractory types and configurations (e.g. Gorog and Adams, 1987c; Adams, 1992; Stiles, 1991). The reduction of emmissions of environmentally detrimental compounds has also gained increased interest in the last few years, mostly due to the need to comply with tightened pollution regulations (Lewis, 1979; Mehra, 1981). Common operational difficulties are listed by Ford and Chen (1991). A common problem is the formation of balls and mud rings inside the kiln, which may result in spill back or kiln cycling (Tran and Barnham, 1990; Tran *et al.*, 1991; Notidis and Tran, 1993). The research associated to energy economy ranges from the determination of industrial configurations that reduce fuel requirements (Mumford *et al.*, 1989; Perry, 1989) to studies of the variables which offer the best potential for decrease in operational costs (Lima *et al.*, 1983). Computer simulation has revealed itself as a powerful tool in the analysis and prediction of the performance of the lime kiln.

Originally, lime quality was determined by visual appearance. Kiln operators would observe the coloration of the lime mud and, based on their experience, adjust the fuel flow to the burner (Ataíde, 1990). Because of decisions based on subjective evaluation, the temperature target would vary from operator to operator and from day to day. To avoid this, computer-based control systems for the lime kiln have proliferated during the last two decades (Uronen *et al.*, 1976; Elsila *et al.*, 1979; Blevins and Rice, 1983; Guimarães *et al.*, 1986; Crowther *et al.*, 1987; Charos *et al.*, 1991; McIlwain, 1992).

A.2 Formulation of Mathematical Models

Simulation of an operating kiln makes it possible to know in advance the effect of variations in the operating parameters, showing the necessary corrections for the improvement of the operation and for the optimization of the kiln for a given production (Guimarães *et al.*, 1986).

Mathematical models for the steady and transient states of cement kilns were first developed in the 60's and 70's (Lyons and *et al.*, 1962; Sass, 1967; Spang, 1972). Because the lime kiln can be thought of as a simplified cement kiln, models for the lime kiln followed (Koivo and Chase, 1972; Pearce, 1973; Dekkiche *et al.*, 1980; Guimarães, 1980; Guimarães and Edwards, 1985; Bailey and Willison, 1985; Gorog and Adams,1987a, 1987b, 1987c, 1987d, 1987e).

The model proposed by Guimarães (1980) has been used extensively for the study in steady-state of the behaviour of the lime kiln (Lima *et al.*, 1983; Edwards and Singh, 1984). In the nineties, two dynamic models were presented by Hsieh *et al.* (1990) and Ribeiro *et al.* (1992). However, these two latest models do not include differential mass balances to the gas phase and therefore cannot be used to predict the instant composition of the exhaust gas. In transient state, significant differences exist between the response time of the gas phase in the chain zone (up to 10 min) and in the combustion zone (very fast) (Guimarães *et al.*, 1986) which is undoubtedly due to the differences in gas convection velocity. This factor is taken into account in the dynamic model developed in this study as local gas convection velocity is evaluated in each integration step as a function of local gas temperature, pressure and composition.

Basically, the assumptions made in the model developed are:

⋆ Mass flowrates of burden and nitrogen are constant throughout the kiln, which is a reasonable assumption for small disturbances in the flowrates.

⋆ No inerts are considered in the solids phase or impurities such as the presence of traces of sodium, silica and potassium compounds. Their major effect in the behaviour of the kiln would be to promote ball or ring formations. Sodium is known to cause sintering of calcium carbonate particles. However, experimental or theoretical correlations do not exist yet.

⋆ Dust losses are considered negligible for the same reason.

⋆ The length of the flame is approximately constant and the rate of the combustion reaction can be described by a linear relationship (Guimarães, 1980).

⋆ The flame is considered to be a homogeneous mixture of air and fuel.

⋆ No pressure gradients are considered throughout the kiln (constant atmospheric pressure).

⋆ The gas phase is assumed to behave as a perfect gas.

⋆ Profiles in the radial and angular coordinates are assumed to be negligible compared to the ones in the axial direction.

⋆ The kiln wall is supposed to be thin enough, compared to the dimensions of the kiln, to be considered as a boundary.

⋆ Heats of reaction and latent heat of vaporization are considered constant, as well as the conductivity of the kiln wall. Solids density and heat capacity are also considered constant.

⋆ The kinetics of the thermal decomposition of calcium carbonate can be represented by an Arrhenius-type expression (Satterfield and Feakes, 1959).

The correlations presented by Spang (1972) and later by Guimarães (1980) for the estimation of heat transfer coefficients were used. A considerable number of correlations has been published and a comparison is made by Edwards and Singh (1984), where it is noted that estimates for the gas-solid heat transfer coefficient can vary as much as four times depending on the correlation used. Thus no accurate estimates can be made *a priori* and the unknown parameters must be adjusted to the case under study.

The equations that describe the dynamic model are listed in Tab.A.1 and the ones corresponding to the steady-state model (Guimarães, 1980) in Tab.A.2. The additional relationships used in both of them are listed in Tab.A.3.

The dynamic model was solved using orthogonal collocation (Villadsen and Michelsen, 1976) in fixed finite elements (Finlayson, 1980) to discretize the spatial coordinate, followed by integration by the BDF method — subroutine LSODI (Hindmarsh, 1980) — using the method of lines (Davis, 1984). The steady-state model was solved using the method proposed by Guimarães (1980) which is basically a shooting method — it is a two point boundary value problem — and consists of arbitrating a profile for the solids, integrate the gas phase variables along z using these values, and then integrate the solids phase along z using the just calculated gas-phase profiles. The profiles of both phases are stored in a number of points ($npoints$) separated by a distance Δz so that when one of the phases is integrated, interpolated values for the other phase in the same z-location can be evaluated. This procedure is repeated until no significant variation occurs for the gas or solids variables between consecutive integrations. Integration along z was also performed using the LSODI subroutine.

A study of the transient behaviour of the kiln for several disturbances has been made and is presented elsewhere (Pais and Portugal, 1993a; Pais and Portugal, 1993b). The mathematical model for the steady-state of the lime kiln has also been used to determine the optimal kiln refractory distribution for a specific set of operating conditions (Pais and Portugal, 1993b).

$$\frac{\partial X_1}{\partial t} = -v_s \frac{\partial X_1}{\partial z} + R_D \tag{A.3}$$

$$\frac{\partial X_2}{\partial t} = -v_s \frac{\partial X_2}{\partial z} - R_D \tag{A.4}$$

$$\frac{\partial X_3}{\partial t} = -v_s \frac{\partial X_3}{\partial z} - R_W \tag{A.5}$$

$$\frac{\partial Y_s}{\partial t} = -v_s \frac{\partial Y_s}{\partial z} + \frac{Q_2 + Q_3}{A_s \rho_s Cp_s T_{s0}} - (Q_W + Q_D) \frac{m_b}{A_s \rho_s Cp_s T_{s0}} \tag{A.6}$$

$$\frac{\partial Y_1}{\partial t} = v_g \frac{\partial Y_1}{\partial z} - v_g \frac{32m + 8n - 16p}{M_F} R_F \tag{A.7}$$

$$\frac{\partial Y_2}{\partial t} = v_g \frac{\partial Y_2}{\partial z} + v_g \frac{44m}{M_F} R_F + \frac{M_{CO2}}{M_{CaO}} R_D \frac{m_b}{m_n} \tag{A.8}$$

$$\frac{\partial Y_3}{\partial t} = v_g \frac{\partial Y_3}{\partial z} + v_g \frac{9n}{M_F} R_F + R_W \frac{m_b}{m_n} \tag{A.9}$$

$$\frac{\partial Y_4}{\partial t} = v_g \frac{\partial Y_4}{\partial z} - v_g R_F \tag{A.10}$$

$$\frac{\partial Y_g}{\partial t} = v_g \frac{\partial Y_g}{\partial z} + \frac{-Q_1 - Q_2 + Q_F}{A_g \rho_g Cp_g T_{g0}} -$$

$$- \left(\frac{M_{CO_2}}{M_{CaO}} R_D \int_{T_s}^{T_g} Cp_{CO_2} dT + R_W \int_{T_s}^{T_g} Cp_{H_2O} dT \right) \frac{m_b}{A_g \rho_g Cp_g T_{g0}} \tag{A.11}$$

Initial conditions:

$$\begin{aligned} &X_1(0,z) = X1_{in}(z), \quad X_2(0,z) = X2_{in}(z), \quad X_3(0,z) = X3_{in}(z), \\ &Y_s(0,z) = Ys_{in}(z), \quad Y_1(0,z) = Y1_{in}(z), \quad Y_2(0,z) = Y2_{in}(z), \\ &Y_3(0,z) = Y3_{in}(z), \quad Y_4(0,z) = Y4_{in}(z), \quad Y_g(0,z) = Yg_{in}(z) \end{aligned} \tag{A.12}$$

Boundary conditions:

$$\begin{aligned} &X_1(t,0) = X1_0(t), \quad X_2(t,0) = X2_0(t), \quad X_3(t,0) = X3_0(t), \\ &Y_s(t,0) = Ys_0(t), \quad Y_1(t,ztot) = Y1_0(t), \quad Y_2(t,ztot) = Y2_0(t), \\ &Y_3(t,ztot) = Y3_0(t), \quad Y_4(t,ztot) = Y4_0(t), \quad Y_g(t,ztot) = Yg_0(t) \end{aligned} \tag{A.13}$$

Table A.1: Differential energy and mass balances for the lime kiln in transient state.

$$\frac{dX_1}{dz} = \frac{R_D}{v_s} \tag{A.14}$$

$$\frac{dX_2}{dz} = -\frac{R_D}{v_s} \tag{A.15}$$

$$\frac{dX_3}{dz} = -\frac{R_W}{v_s} \tag{A.16}$$

$$\frac{dY_s}{dz} = \frac{Q_2 + Q_3}{A_s \rho_s v_s Cp_s T_{s0}} - (Q_W + Q_D)\frac{m_b}{A_s \rho_s v_s Cp_s T_{s0}} \tag{A.17}$$

$$\frac{dY_1}{dz} = \frac{32m + 8n - 16p}{M_F} R_F \tag{A.18}$$

$$\frac{dY_2}{dz} = -\frac{44m}{M_F} R_F - \frac{M_{CO_2}}{M_{CaO}} \frac{R_D}{v_g} \frac{m_b}{m_n} \tag{A.19}$$

$$\frac{dY_3}{dz} = -\frac{9n}{M_F} R_F - \frac{R_W}{v_g} \frac{m_b}{m_n} \tag{A.20}$$

$$\frac{dY_4}{dz} = R_F \tag{A.21}$$

$$\frac{dY_g}{dz} = \frac{Q_1 + Q_2 - Q_F}{A_g \rho_g Cp_g v_g T_{g0}} +$$

$$+\left(\frac{M_{CO_2}}{M_{CaO}} R_D \int_{T_s}^{T_g} Cp_{O_2} dT + R_W \int_{T_s}^{T_g} Cp_{H2O} dT\right) \frac{m_b}{A_g \rho_g Cp_g v_g T_{g0}} \tag{A.22}$$

Boundary conditions:

$$\begin{aligned} X_1(z=0) &= X1_0, & X_2(z=0) &= X2_0, & X_3(z=0) &= X3_0, \\ Y_s(z=0) &= Ys_0, & Y_1(z=ztot) &= Y1_0, & Y_2(z=ztot) &= Y2_0, \\ Y_3(z=ztot) &= Y3_0, & Y_4(z=ztot) &= Y4_0, & Y_g(z=ztot) &= Yg_0 \end{aligned} \tag{A.23}$$

Table A.2: Differential energy and mass balances for the lime kiln in steady state.

$$
\begin{array}{lcl@{\qquad}lcl}
A_g & = & \pi\frac{di^2}{4} - A_s & Q_1 & = & A_1 h_1 (T_g - T_w) \\
A_s & = & \frac{W_s}{\rho_s v_s} & Q_2 & = & A_2 h_2 (T_g - T_s) \\
A_1 & = & \pi di(1 - \frac{\theta}{360}) & Q_3 & = & A_3 h_3 (T_w - T_s) \\
A_2 & = & di\ sin(\pi\frac{\theta}{360}) & Q_4 & = & A_4 h_4 (T_w - T_{wp}) \\
A_3 & = & \pi di \frac{\theta}{360} & Q_5 & = & A_5 h_5 (T_{wp} - T_a) \\
A_4 & = & \pi\frac{(do-di)}{ln\frac{do}{di}} & Q_1 & = & Q_3 + Q_4 \\
A_5 & = & \pi do & Q_4 & = & Q_5
\end{array}
$$

$$
\begin{array}{lcl}
Cp_g & = & \dfrac{\frac{Cp_{N_2}}{M_{N_2}} + \frac{Y_1 Cp_{O_2}}{M_{O2}} + \frac{Y_2 Cp_{CO_2}}{M_{CO_2}} + \frac{Y_3 Cp_{H_2O}}{M_{H_2O}} + \frac{Y_4 Cp_F}{M_F}}{1+Y_1+Y_2+Y_3+Y_4} \times 10^3 \\
Cp_i & = & a_i + b_i T + c_i t^2 + d_i t^3 \\
K_D & = & A_C exp(-\Delta E_D / RT_s) \\
K_W & = & A_W exp(-\Delta E_W / RT_s) \\
h_1 & = & f_1 + 5.73 \times 10^{-8} e_g e_w (T_g^3 + T_g^2 T_w + T_g T_w^2 + T_w^3) \\
h_2 & = & f_2 + 5.73 \times 10^{-8} e_g e_s (T_g^3 + T_g^2 T_s + T_g T_s^2 + T_s^3) \\
h_3 & = & f_3 + 5.73 \times 10^{-8} e_w e_s (T_s^3 + T_s^2 T_w + T_s T_w^2 + T_w^3) \\
h_4 & = & \frac{2K}{do-di} \\
h_5 & = & f_5 + 5.73 \times 10^{-8} e_{wp} (T_{wp}^3 + T_{wp}^2 T_a + T_{wp} T_a^2 + T_a^3)
\end{array}
$$

$$
\begin{array}{lcl@{\qquad}lcl}
m_b & = & \frac{W_s}{1+X_2\frac{M_{CO_2}}{M_{CaO}}+X_3}\frac{1}{v_s} & & & \\
m_n & = & \frac{W_g}{1+Y_1+Y_2+Y_3+Y_4}\frac{1}{v_g} & zm_{met} & = & \frac{z_{met} M_{CH_4}}{z_{met} M_{CH_4} + z_{et} M_{C_2H_6} + z_{prop} M_{C_3H_8}} \\
n_v & = & \frac{P_a}{RT_g} & & & \\
Q_W & = & R_W \Delta H_W & zm_{et} & = & \frac{z_{et} M_{C_2H_6}}{z_{met} M_{CH_4} + z_{et} M_{C_2H_6} + z_{prop} M_{C_3H_8}} \\
Q_D & = & R_D \Delta H_C & & & \\
Q_F & = & R_F \Delta H_F W_{gn} & zm_{prop} & = & \frac{z_{prop} M_{C_3H_8}}{z_{met} M_{CH_4} + z_{et} M_{C_2H_6} + z_{prop} M_{C_3H_8}} \\
R_D & = & K_D X_2 & & & \\
R_F & = & \frac{Y_4}{z-z_5} & \Delta H_F & = & zm_{met} \Delta H f_{met} + zm_{et} \Delta H f_{et} + \\
 & & & & + & zm_{prop} \Delta H f_{prop}
\end{array}
$$

$$
\begin{array}{lcl}
R_W & = & \begin{cases} K_W \text{ if } X_3 \geq 0.1 \\ 10 K_W X_3 \text{ if } X_3 < 0.1 \end{cases} \\
v_g & = & \frac{W_g}{A_g \rho_g} \\
v_s & = & 0.054\frac{n_r S di}{(\phi+24)} \\
\rho_g & = & \dfrac{1+Y_1+Y_2+Y_3+Y_4}{\frac{1}{M_{N2}}+\frac{Y_1}{M_{O2}}+\frac{Y_2}{M_{CO_2}}+\frac{Y_3}{M_{H2O}}+\frac{Y_4}{M_F}} n_v \times 10^{-3} \\
W_s & = & (1 + X_2\frac{M_{CO_2}}{M_{CaO}} + X_3) W_b \quad W_g = W_{gn}(1 + Y_1 + Y_2 + Y_3 + Y_4)
\end{array}
$$

Table A.3: Additional relationships for the lime kiln models.

Notation

A_1	=	Area of contact gas-wall per unit length of kiln (m^2/m)
A_2	=	Area of contact gas-solid per unit length of kiln (m^2/m)
A_3	=	Area of contact solid-wall per unit length of kiln (m^2/m)
A_4	=	Area for heat transfer across wall per unit length of kiln (m^2/m)
A_5	=	Area of contact outer wall - ambient air per unit length of kiln (m^2/m)
A_C	=	Arrhenius factor for the decomposition reaction (s^{-1})
A_s	=	Cross sectional area of the kiln occupied by solids (m^2)
A_g	=	Cross sectional area of the kiln occupied by gas (m^2)
A_W	=	Arrhenius factor for the evaporation of water (s^{-1})
Cp_g	=	Heat capacity of the gas phase (J/KgK)
Cp_i	=	Heat capacity of gas component i ($J/molK$)
Cp_s	=	Heat capacity of the solids phase (J/KgK)
di	=	Inner diameter of the kiln (m)
d_1	=	Inner diameter of the kiln in zone 1 (m)
d_2	=	Inner diameter of the kiln in zone 2 (m)
d_3	=	Inner diameter of the kiln in zone 3 (m)
do	=	Outer diameter of the kiln (m)
e_g	=	Gas emissivity factor
e_s	=	Solids emissivity factor
e_w	=	Inner wall emissivity factor
e_{wp}	=	Outer wall emissivity factor
f_1	=	Parameter in expression for h_1 (J/sm^2K)
f_2	=	Parameter in expression for h_2 (J/sm^2K)
f_3	=	Parameter in expression for h_3 (J/sm^2K)
f_5	=	Parameter in expression for h_5 (J/sm^2K)
h_1	=	Heat transfer coefficient gas-wall (J/sm^2K)
h_2	=	Heat transfer coefficient gas-solid (J/sm^2K)
h_3	=	Heat transfer coefficient solid-wall (J/sm^2K)
h_4	=	Heat transfer coefficient inner-outer wall (J/sm^2K)
h_5	=	Heat transfer coefficient outer wall - ambient air (J/sm^2K)
K	=	Thermal conductivity of kiln wall (J/smK)
K_D	=	Kinetic parameter for the decomposition reaction (s^{-1})
K_W	=	Kinetic parameter for the evaporation of water from the solids phase (s^{-1})
m_b	=	Mass of burden per unit length of kiln (Kg/m)
m_n	=	Mass of N_2 / unit length of kiln (Kg/m)
m, n and p	=	Parameters of the general formula for fuel $C_mH_nO_p$
M_{CaO}	=	Molecular mass of CaO (g/mol)
M_{CO_2}	=	Molecular mass of CO_2 (g/mol)
M_{CH_4}	=	Molecular mass of CH_4 (g/mol)
$M_{C_2H_6}$	=	Molecular mass of C_2H_6 (g/mol)
$M_{C_3H_8}$	=	Molecular mass of C_3H_8 (g/mol)
M_{H_2O}	=	Molecular mass of H_2O (g/mol)
M_{N_2}	=	Molecular mass of N_2 (g/mol)
M_{O_2}	=	Molecular Mass of O_2 (g/mol)
M_F	=	Molecular mass of fuel (g/mol)
$npoints$	=	Number of points where the gas and solids profiles are stored
n_v	=	Number of moles per m^3 (mol/m^3)
n_r	=	Rotation speed of kiln (Rpm) in the steady state model

P_a	=	Atmospheric Pressure (Pa)
Q_1	=	Heat transferred from gas to wall (J/ms)
Q_2	=	Heat transferred from gas to solids (J/ms)
Q_3	=	Heat transferred from wall to solids (J/ms)
Q_4	=	Heat transferred from inner to outer wall ($J/m/s$)
Q_5	=	Heat lost to the surroundings (ambient air) (J/ms)
Q_D	=	Heat consumed by the decomposition reaction (J/Kgs)
Q_F	=	Heat generated by the combustion of fuel (J/ms)
Q_W	=	Heat consumed by the evaporation of water (J/Kgs)
R	=	Perfect gas constant ($J/molK$)
R_D	=	Rate of decomposition reaction (Kg/Kgs)
R_W	=	Rate of the evaporation of water (Kg/Kgs)
R_F	=	Rate of the combustion of fuel (Kg/Kgm)
S	=	Inclination of the kiln (%)
t	=	Time (s)
T_a	=	Ambient temperature (K)
T_g	=	Gas temperature (K)
T_s	=	Solids temperature (K)
T_{g0}	=	Adimensionalization constant for the gas temperature (K)
T_{s0}	=	Adimensionalization constant for the solids temperature (K)
T_w	=	Inner wall temperature (K)
T_{wp}	=	Outer wall temperature (K)
v_s	=	Solids convection velocity (m/s)
v_g	=	Gas convection velocity (m/s)
X_1	=	Kg free CaO/Kg burden (Kg/Kg) (burden = free CaO + combined CaO)
X_2	=	Kg combined CaO/Kg burden (Kg/Kg)
X_3	=	Kg H_2O in solids phase $/Kg$ burden (Kg/Kg)
$X1_{in}, X2_{in}, X3_{in}$	=	Initial conditions for X_1, X_2 and X_3
$X1_0, X2_0, X3_0$	=	Boundary conditions for X_1, X_2 and X_3
Ys	=	Ts/T_{s0} (K/K)
Y_1	=	Kg $O2/KgN2$ in gas stream
Y_2	=	Kg $CO2/KgN2$ in gas stream
Y_3	=	Kg $H2O/KgN2$ in gas stream
Y_4	=	Kg Fuel $/KgN2$ in gas stream
Y_g	=	Tg/T_{g0}
$Ys_{in}, Y1_{in}, Y2_{in}, Y3_{in}, Y4_{in}, Yg_{in}$	=	Initial conditions for Y_s, Y_1, Y_2, Y_3, Y_4 and Y_g
$Ys_0, Y1_0, Y2_0, Y3_0, Y4_0, Yg_0$	=	Boundary conditions for Y_s, Y_1, Y_2, Y_3, Y_4 and Y_g
W_b	=	Burden mass flowrate (Kg/s)
W_s	=	Total solids mass flowrate (Kg/s)
W_g	=	Total gas mass flowrate (Kg/s)
W_{gn}	=	Nitrogen mass flowrate (Kg/s)
z	=	Distance from solids feed end (m)
$zd1$	=	Location where refractory zone 1 ends (measured from solids feed end) (m)
$zd2$	=	Location where refractory zone 2 ends (measured from solids feed end) (m)

z_{et}	=	Molar fraction of ethane in fuel
z_{met}	=	Molar fraction of methane in fuel
z_{prop}	=	Molar fraction of propane in fuel
zm_{met}	=	Mass fraction of methane in fuel
zm_{et}	=	Mass fraction of ethane in fuel
zm_{prop}	=	Mass fraction of propane in fuel
z_{tot}	=	Total length of kiln (m)
z_5	=	Start of flame (distance from solids feed end) (m)

Greek Letters

ΔE_D	=	Activation energy for the decomposition reaction (J/mol)
ΔE_W	=	Activation energy for the evaporation of water (J/mol)
ΔH_C	=	Heat of reaction for the decomposition reaction (J/Kg)
ΔH_F	=	Heat of the combustion of fuel (J/Kg)
ΔHf_{met}	=	Combustion heat of methane (J/Kg)
ΔHf_{et}	=	Combustion heat of ethane (J/Kg)
ΔHf_{prop}	=	Combustion heat of propane (J/Kg)
ΔH_W	=	Latent heat of vaporization of water (J/Kg)
Δz	=	Distance between points in the steady state model numerical resolution (m)
ρ_g	=	Density of the gas phase (Kg/m^3)
ρ_s	=	Density of the solids phase (Kg/m^3)
θ	=	Fill angle of the solids (^{o})
ϕ	=	Angle of repose of solids (^{o})

Appendix B

The Causticizing Battery

B.1 Introduction

The slaking and causticizing operations are a fundamental part of the chemical recovery process in kraft paper pulp mills. They are carried out in a series of continuous stirred tank reactors (see Fig.B.1). The reburned lime from the kiln

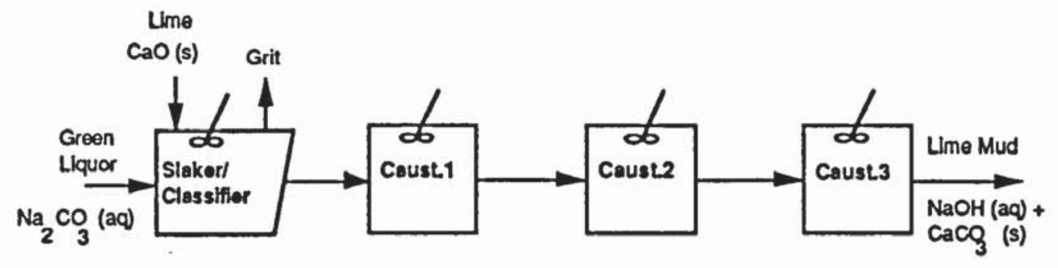


Figure B.1: Schematic representation of the causticizing battery.

is conveyed into the slaker at a rate adjusted to suit the amount of sodium carbonate present in the green liquor. The main chemical reactions that take place are the reaction of quicklime with water to form calcium hydroxide (slaking reaction)

$$CaO(s) + H_2O(l) \xrightarrow{R_1} Ca(OH)_2(s) + (-\Delta H_s) \tag{B.1}$$

and the reaction of calcium hydroxide with sodium carbonate to form sodium hydroxide (causticizing reaction):

$$Ca(OH)_2(s) + Na_2CO_3(aq) \underset{R_3}{\overset{R_2}{\rightleftharpoons}} CaCO_3(s) + 2NaOH(aq) + (-\Delta H_c) \tag{B.2}$$

Although the values for the heat of reaction found in the literature differ from author to author (Kojo, 1979b; Mehra *et al.*, 1985; Dorris and Allen, 1986; Leiviska *et al.*, 1987) the slaking reaction is highly exothermic and the causticizing reaction is normally taken to be nearly athermic. Especially in the slaker, evaporation of water takes place due to the rise in temperature due to heat generation, according to

$$H_2O(l) \rightarrow H_2O(g) + (-\Delta H_v) \tag{B.3}$$

No other reactions are considered in this study. In the slaker, both the hydration of calcium oxide and the formation of calcium carbonate take place simultaneously. Lime can normally be considered totally slaked when it leaves the slaker, and in most cases only the causticizing reaction takes place in the causticizing tanks. The lime slurry is classified in the slaker to remove sand, stone, improperly burned lime and larger particles (grit). Maximum conversion for the causticizing reaction requires additional agitation of about one and a half hours after the slaker, and three or four agitated causticizing tanks are generally employed in series.

Advanced control systems provide automatic responses to lime quality variations, changes in green liquor composition and operating rate adjustments. However, the influence of common disturbances on key measurements must be understood (Mittet, 1994) and, if possible, quantitatively predicted. Computer simulation is invaluable for the achievement of both these objectives.

In this study, a microscopic model for the solid particles undergoing the causticizing reaction was developed and applied to the determination of the steady state of the causticizing units (Portugal and Pais, 1989; Pais and Portugal, 1994a; Pais and Portugal, 1994b). An approximate pseudo-homogeneous dynamic model was also developed and implemented for the transient state of the system (Pais and Portugal, 1994c).

B.2 Formulation of Mathematical Models — Heterogeneous Approach

B.2.1 Mathematical Model for the Solid Particles

Heterogeneous fluid-solid reactions are usually influenced to a high degree by heat and mass transfer processes (Doraiswamy and Sharma, 1984). A general heterogeneous reaction, where reactants and products can be either in the solid or fluid phases, can be represented by

$$\nu_A A(f) + \nu_R R(s) \rightleftharpoons \nu_B B(f) + \nu_S S(s) \tag{B.4}$$

In the case of the causticizing reaction, liquid reactant A corresponds to CO_3^{2-}, liquid product B to OH^-, solid reactant R to $Ca(OH)_2$ and solid product S to $CaCO_3$.

The causticizing reaction (Angevine, 1983; Dorris and Allen, 1985; Dorris and Allen, 1986; Blackwell, 1987) is an example of a non-catalytic fluid-solid reaction. Two basic types of models have been considered so far for non-catalytic fluid-solid reactions: the sharp interface model, (SIM) (Yagi and Kunii, 1955, Levenspiel, 1972) and the homogeneous model. Several extensions to the basic SIM have been proposed (Park and Levenspiel, 1975; Park and Levenspiel, 1977; Georgakis *et al.*, 1979; Ranade and Harrison, 1979). In the homogeneous model, the fluid penetrates deeply into the solid and reaction takes place throughout the particle. Comparison between these two models has been made by several authors (Ishida and Wen, 1968; Ramachandran and Doraiswamy, 1982; Doraiswamy and Sharma, 1984).

Earlier authors (Rydin, 1978; Kojo, 1979b; Kojo, 1980) suggested dissolution-reaction-precipitation mechanisms for the slaking and causticizing reactions. Nowadays, it is generally accepted that they proceed via a heterogeneous mechanism at the solid-liquid interface (Angevine, 1983; Dorris and Allen, 1985; Dorris and Allen, 1986; Blackwell, 1987; Dorris, 1993). Also, the causticizing reaction is described as being strongly dependent on internal diffusion limitations. A study of the mechanism of the slaking reaction has been made by Dutta and Shirai (1980).

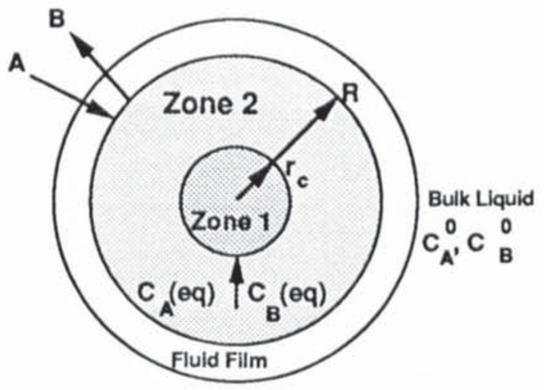


Figure B.2: Schematic Representation of the SIM model.

No studies have been found in the literature concerning the possibility of a sharp interface model when the reaction is reversible, which is the case of the causticizing reaction. An analysis of the applicability of the sharp interface model to this case is made elsewhere (Pais and Portugal, 1994b; Portugal and Pais, 1989). It is shown that, under certain circumstances, a sharp interface may exist for reversible reactions.

In this work, a sharp interface model has been developed for reactions of the type described by eq.(B.4) for cases (such as most liquid-solid reactions) where the pseudo-steady approximation is not valid. The model relies on the assumptions that mass transfer (either through the ash layer or fluid film) is the limiting step, reversible reaction is instantaneous, and spherical particle is isothermal. If the rate of both forward and reverse reactions is high compared to the rate of mass transfer, then chemical equilibrium exists at the reaction interface. Physical properties of the solid are allowed to be different in the inner shrinking core (zone 1) and ash layer (zone 2), as well as effective diffusion coefficients for liquid species A and B in both zones. Fig.B.2 represents graphically the classical SIM pattern, where zone 1 corresponds to the inner shrinking core and Zone 2 to the ash layer. If the exponents in the equilibrium relationship are taken to be equal to the stoichiometric coefficients in the reversible reaction, then the model is described by the set of partial differential equations presented in Table B.1, where r_c represents the position of the reaction interface.

Eqs.(B.9) and (B.10) state that:

a) for a certain amount of reacted A, a corresponding amount of B is produced, according to their stoichoimetric relationship;

b) concentrations at the reaction interface obey to the equilibrium relationship.

Together with eq.(B.6), they imply that the rate of advance of the reaction front is such that the concentrations at the interface, momentarily disturbed by the diffusion process, return to their equilibrium values.

After the reaction front has reached the center of the particle, only one zone will remain, where diffusion alone occurs. For numerical purposes, this happens as soon as r_c becomes lower than an arbitrary value so that the two zones still

$$\frac{\partial C}{\partial t} = \frac{De}{\varepsilon}\left(\frac{\partial^2 C}{\partial r^2} + \frac{2}{r}\frac{\partial C}{\partial r}\right) \tag{B.5}$$

$0 < r < r_c(De = De_1, \varepsilon = \varepsilon_1)$ and $r_c < r < R(De = De_2, \varepsilon = \varepsilon_2), C = C_A, C_B$

$$\frac{dr_c}{dt} = -\frac{1}{C_R^{in}}\frac{\nu_R}{\nu_A}\left[De_{A2}\left.\frac{\partial C_A}{\partial r}\right)_{r_c^+} - De_{A1}\left.\frac{\partial C_A}{\partial r}\right)_{r_c^-}\right] \tag{B.6}$$

Boundary conditions:

$$\frac{\partial C}{\partial r} = 0 \quad r = 0, C = C_A, C_B \tag{B.7}$$

$$K_L(C^0 - C) = De_2\frac{\partial C}{\partial r} \quad r = R, C = C_A, C_B \tag{B.8}$$

$$De_{A2}\left.\frac{\partial C_A}{\partial r}\right)_{r_c^+} - De_{A1}\left.\frac{\partial C_A}{\partial r}\right)_{r_c^-} = -\frac{\nu_A}{\nu_B}\left[De_{B2}\left.\frac{\partial C_B}{\partial r}\right)_{r_c^+} - De_{B1}\left.\frac{\partial C_B}{\partial r}\right)_{r_c^-}\right] \tag{B.9}$$

$r = r_c$

$$\frac{C_B^{\nu_B}}{C_A^{\nu_A}} = K_E \quad r = r_c \tag{B.10}$$

Initial conditions:

$$C = C^{in} \quad t = 0, C = C_A, C_B \tag{B.11}$$

$$r_c = r_c^{in} \quad t = 0 \tag{B.12}$$

Table B.1: Microscopic model for the solid particles in the causticizing tanks.

remain. When this is the case, then equality of mass fluxes must be specified on both sides of the interface and eqs.(B.9) and (B.10) are replaced by (B.13) and (B.14):

$$De_{A2}\left.\frac{\partial C_A}{\partial r}\right)_{r_c^+} = De_{A1}\left.\frac{\partial C_A}{\partial r}\right)_{r_c^-} \qquad r = r_c \tag{B.13}$$

$$De_{B2}\left.\frac{\partial C_B}{\partial r}\right)_{r_c^+} = De_{B1}\left.\frac{\partial C_B}{\partial r}\right)_{r_c^-} \qquad r = r_c \tag{B.14}$$

Table B.1 (cont.): Microscopic model for the solid particles in the causticizing tanks.

As a consequence, the reaction interface ceases to advance.

The model was solved using orthogonal collocation (Villadsen and Michelsen, 1976) in both the unreacted core and the ash layer, using eqs.(B.9) and (B.10) or (B.13) and (B.14) to couple the two zones. The resulting set of ODE's was then solved by the BDF method using subroutine LSODI (Hindmarsh, 1980). In order to immobilize the moving boundary, a change of variables similar to the one used by Stamatakis and Tien (1991) was made. A detailed description of the solution process is given elsewhere (Pais and Portugal, 1994b). The effect of reversibility is similar to a decrease in the driving force for diffusion due to an increase in the concentration of A at the reaction front compared to the irreversible case.

B.2.2 Steady State Model for the Causticizing Units

Information concerning the behaviour of non-catalytic heterogeneous CSTR's is scarce; the exact description of these systems is very complex for steady state as, even if the concentrations of the liquid phase are constant, the solids phase will always be in transient state. In this study, the macroscopic model for the CSTR has been coupled to the microscopic model for the solid particles presented above. Some of the assumptions used by Tsai and Tsao (1991) are also valid for this model:

— fluid is perfectly mixed and the solid is homogeneously suspended so that there are no concentration gradients in the bulk liquid;

— system is isothermal.

However, in the case under study the concentration at the surface of a certain particle is a function of both its age t and diameter D ($R = D/2$) and therefore there are concentration differences at the surfaces of the solid particles. If the residence time distribution of the solid particles in the reactor is given by $e(t)$, and the distribution of diameters in the solids feed is given by $f(D)$, the infinitesimal number of particles of age t and diameter D (per slurry unit volume) is $\frac{\alpha f(D) e(t)}{\frac{4}{3}\pi\left(\frac{D}{2}\right)^3} dD\, dt$. If the particles are spherical, each has an outer superficial area of $4\pi\left(\frac{D}{2}\right)^2$. The rate of disappearance of reactant A from the bulk liquid per slurry unit volume, here designated by r_A, is thus

$$r_A = \int_{t=0}^{+\infty} \int_{D=0}^{+\infty} 6e(t)\frac{\alpha}{D} f(D) K_{L_A}(C_A^0 - C_{A_{r=R}}(D,t)) dD\ dt \tag{B.15}$$

Although r_A is in fact a mass transfer rate, it will henceforth be referred to as the reaction rate, since for practical purposes it represents an *apparent* reaction rate. For liquid-phase product B we have ($r_B < 0$)

$$r_B = \int_{t=0}^{+\infty} \int_{D=0}^{+\infty} 6e(t)\frac{\alpha}{D} f(D) K_{L_B}(C_B^0 - C_{B_{r=R}}(D,t)) dD\ dt \tag{B.16}$$

If the CSTR is assumed to be ideally mixed, and supposing that the residence time distribution is similar in both phases, which is a reasonable assumption for the causticizing system (Hypponen and Luukko, 1984), we have $e(t) = \frac{Q}{V} e^{-\frac{Qt}{V}}$.

In steady state, then

$$Q(1-\alpha)C_A^i - Q(1-\alpha)C_A^0 = r_A V \tag{B.17}$$

and similarly for product B,

$$Q(1-\alpha)C_B^i - Q(1-\alpha)C_B^0 = r_B V \tag{B.18}$$

Equations (B.5–B.8), (B.9–B.10) or (B.13–B.14), and (B.15–B.18) must be solved simultaneously in order to determine C_A^0 and C_B^0, the steady state bulk concentrations of the liquid phase of the reactor. D is normally used, rather than R, in size distribution functions, explaining its use here. The full set of equations that describe the particles must be solved for all different combinations of ages and diameters (theoretically, from $D = 0$ to $D = +\infty$ and from $t = 0$ to $t = +\infty$, although in practice only limited ranges need be considered).

After the solution is found, the mean volumetric solids conversion can be readily evaluated (Levenspiel, 1972):

$$\overline{x} = \int_{t=0}^{+\infty} \int_{D=0}^{+\infty} e(t) f(D) x(D,t) dD\ dt \tag{B.19}$$

A detailed description of the iterative procedure used to solve the equations presented here is given elsewhere (Pais and Portugal, 1994a), as well as a study of the effect of variation of the parameters that describe the model. This model makes it possible to evaluate the steady state concentrations for two-phase continuous stirred tank reactors, based solely on the characteristics of the particles and operational parameters. However, the computational solution is highly time-consuming due to the very high number of discretization points for the residence time distribution that must be used (for some cases tried, the order of magnitude was of hundreds of thousands).

Notation

C	=	Concentration (mol/m^3 liquid) except C_R^{in} (mol/m^3 solid)
D	=	Particle diameter ($D = 2R$) (m)
De	=	Effective diffusion coefficient (m^2/s)
$e(t)$	=	Residence time distribution function
$E(t)$	=	Cumulative residence time function
$f(D)$	=	Diameter distribution function
K_E	=	Equilibrium constant ($(mol/m^3)^{\nu_B}/(mol/m^3)^{\nu_A}$)
K_L	=	Mass transfer coefficient (m/s)
r	=	Spatial radial coordinate (m) (except r_A, r_B and r_c)
r_c	=	Location of the reaction front (m)
r_A	=	Rate of disappearance of A from the bulk liquid referred to unit volume of slurry (mol/m^3s)
r_B	=	Rate of disappearance of B from the bulk liquid referred to unit volume of slurry (mol/m^3s)
R	=	Outer radius of the particle (m)
t	=	Age of the particle (s)
x	=	Volumetric conversion of particle
$\bar{x}$	=	Mean volumetric conversion of particles inside the reactor

Greek letters

α	=	Volumetric fraction of solids inside reactor (equal, in steady state, to the fraction in the feed slurry)
ε	=	Porosity of the particle
ν	=	Stoichiometric coefficient in reaction

Superscripts

i	=	Referred to the inlet of the reactor
in	=	Referred to initial conditions
0	=	Referred to bulk concentration

Subscripts

A	=	Referred to species A
B	=	Referred to species B
c	=	Referred to the reaction front
R	=	Referred to species R
S	=	Referred to species S
1	=	Referred to inner zone (unreacted core of particle)
2	=	Referred to outer zone (ash layer of particle)

B.3 Dynamic Model for the Slaking and Causticizing Units — Pseudo-homogeneous Approach

In recent years, a considerable number of experimental studies have been carried out in order to determine the kinetic laws that govern the slaking and causticizing reactions (Rydin, 1978; Turc *et al.*, 1982; Lindberg and Ulmgren, 1986; Dotson and Krishnagopalan, 1990). The type of lime has a determinant influence on the reaction rates, giving rise to different values for the Arrhenius constants and activation energies. Experimental studies have also obtained values and

	Solution containing Na_2CO_3 only initially	Synthetic green liquor
$\ln(k_2')$	$-\frac{29.0}{RT} + 4.25$	$-\frac{37.0}{RT} + 6.66$
$\ln(k_3')$	$-\frac{25.6}{RT} - 8.30$	$-\frac{28.0}{RT} - 7.25$

Table B.2: Kinetic parameters for the causticizing reaction. (Dotson and Krishnagopalan, 1990).

correlations for the apparent equilibrium constant as a function of concentration (Lindberg and Ulmgren, 1983; Lindberg and Ulmgren, 1986; Dorris, 1990).

In this study, the rate of reaction for the slaking reaction was assumed to be given by the following expression (Rydin, 1978; Turc *et al.*, 1982):

$$R_{CaO} = -k_1 C_{CaO} \qquad \text{(B.20)}$$

where (Turc *et al.*, 1982):

$$ln(60k_1) = -\frac{8645}{T} + 19.72 \text{ (90\% of causticizable } CaO) \qquad \text{(B.21)}$$

(The factor 60 is necessary to convert k_1, originally expressed in min^{-1}, to s^{-1}).

The kinetic expression used for the rate of the causticizing reaction was (Dotson and Krishnagopalan, 1990):

$$R_{Na_2CO_3} = [Na_2CO_3]_o \left(k_2'[Na_2CO_3]^a - k_3'[NaOH]^b\right) \qquad \text{(B.22)}$$

where $[Na_2CO_3]_o$ is the initial sodium carbonate concentration of green liquor. Optimal adjustment was found to correspond to $a = 1$ and $b = 3$. This expression corresponds to $R_2 = [Na_2CO_3]_o k_2'[Na_2CO_3]^a$ and
$R_3 = [Na_2CO_3]_o k_3'[NaOH]^b$. $[Na_2CO_3]$ and $[NaOH]$ are expressed in gNa_2O/l.

In this work, exps.(B.20) and (B.21) were used for the evaluation of the rate of the slaking reaction. Exp.(B.22), together with the parameters obtained by Dotson and Krishnagopalan (1990) (see Table B.2) for synthetic green liquor, were used for the evaluation of the rate of the causticizing reaction. Note that an adequate unit conversion (to S.I. units) must be done before k_2' and k_3' can be used in the model.

The model for the slaking and causticizing units is based on the model proposed by Galtung and Williams (1969), although these authors presented no simulation results. Jacobi and Williams (1973a, 1973b) later chose not to use the Galtung and Williams model due to its complexity and used a more approximate model. All concentrations are related to the slurry unit volume. The equations that describe the behaviour of the slaker are presented in Table B.3.

Note that $R_{CaO} = R_{H_2O} = R_1$, $R_{CO_3^=} = R_2 - R_3$, $R_{OH^-} = 2(R_3 - R_2)$, $R_{CaCO_3} = R_3 - R_2$ and $R_{Ca(OH)_2} = -R_1 + R_2 - R_3$. Similar equations hold for the causticizing tanks, with the relevant changes, and will not be repeated

$$\frac{d[V_1 C_{CO_3^{2-}}{}_1]}{dt} = Q_o C_{CO_3^{2-}}{}_o - Q_1 C_{CO_3^{2-}}{}_1 - R_{CO_3^{2-}}{}_1 V_1 \tag{B.23}$$

$$\frac{d[V_1 C_{OH^-1}]}{dt} = Q_o C_{OH^-}{}_o - Q_1 C_{OH^-1} - R_{OH^-1} V_1 \tag{B.24}$$

$$\frac{d[V_1 C_{H_2O1}]}{dt} = Q_o C_{H_2O_o} - Q_1 C_{H_2O1} - R_{H_2O1} V_1 - Ev_{H_2O1} \tag{B.25}$$

$$\frac{d[V_1 C_{CaO1}]}{dt} = F_{CaO}(1-\beta) - Q_1 C_{CaO1} - R_{CaO1} V_1 \tag{B.26}$$

$$\frac{d[V_1 C_{Ca(OH)_21}]}{dt} = -Q_1 C_{Ca(OH)_21} - R_{Ca(OH_2)_1} V_1 \tag{B.27}$$

$$\frac{d[V_1 C_{CaCO_31}]}{dt} = F_{CaCO_3}(1-\beta) - Q_1 C_{CaCO_31} - R_{CaCO_31} V_1 \tag{B.28}$$

$$\begin{aligned}\frac{dV_1}{dt} &= Q_o + F_{CaO}(1-\beta)\frac{PM_{CaO}}{\rho_{CaO}} + F_{CaCO_3}(1-\beta)\frac{PM_{CaCO_3}}{\rho_{CaCO_3}} - \\ &-Q_1 - Ev_{H2O1}\frac{PM_{H_2O}}{\rho_{H2O}}\end{aligned} \tag{B.29}$$

$$\begin{aligned}\rho_1 Cp_1 \frac{d[V_1 T_1]}{dt} &= Q_o Cp_o \rho_o T_o + \\ &+F_{CaO} Cp_{CaO} PM_{CaO} T_{CaO} + F_{CaCO_3} Cp_{CaCO_3} PM_{CaCO_3} T_{Cao} - \\ &-\beta F_{CaO} Cp_{CaO} PM_{CaO} T_1 - \beta F_{CaCO_3} Cp_{CaCO_3} PM_{CaCO_3} T_1 - \\ &-(R_1 \Delta H_s + (R_2 - R_3)\Delta H_c) V_1 - Ev_{H_2O1} \Delta H_v - \\ &-Ev_{H2O} Cp_{H_2O} PM_{H_2O} T_1 - Q_1 Cp_1 \rho_1 T_1 - q_1\end{aligned} \tag{B.30}$$

Initial conditions:

$$\begin{aligned}
C_{CO_3^{2-}}{}_1(t=0) &= C^{in}_{CO_3^{2-}}{}_1 && \text{(B.31)}\\
C_{OH^-1}(t=0) &= C^{in}_{OH^-1} && \text{(B.32)}\\
C_{H_2O1}(t=0) &= C^{in}_{H_2O1} && \text{(B.33)}\\
C_{CaO1}(t=0) &= C^{in}_{CaO1} && \text{(B.34)}\\
C_{Ca(OH)_21}(t=0) &= C^{in}_{Ca(OH)_21} && \text{(B.35)}\\
C_{CaCO_31}(t=0) &= C^{in}_{CaCO_31} && \text{(B.36)}\\
V_1(t=0) &= V_1^{in} && \text{(B.37)}\\
T_1(t=0) &= T_1^{in} && \text{(B.38)}\\
& && \text{(B.39)}
\end{aligned}$$

Table B.3: Pseudo-homogeneous model for the slaking and causticizing tanks.

here (F_{CaO} and F_{CaCO_3} will be replaced by the incoming flow from the previous tank).

The specific system considered here is composed of one slaker and three causticizers and removal of grit takes place in the slaker only.

Each tank consists of a set of eight variables dependent on time. All the differential equations were integrated simultaneously which corresponds to a set of thirty two ODE's. The equations were integrated using the LSODI integrator (Hindmarsh, 1980).

B.3.1 Additional Assumptions

Evaluation of the Amount of Evaporated Water

If the temperature reaches the boiling point, an estimate for the amount of vaporized water can be made assuming that the boiling temperature is never exceeded.

Taking eqs.(B.29) and (B.30), and assuming that $\frac{dT_1}{dt} = 0$, it is possible to replace the expression for $\frac{dV_1}{dt}$ in eq.(B.30) and obtain an algebraic equation which involves the amount of evaporated water, Ev_{H2O}, so that the temperature balance yields a variation of zero in T_1. Solving for the amount of water actually evaporated, we obtain

$$
\begin{aligned}
Ev_{H_2O} = {} & \frac{1}{\left(-\rho_1 Cp_1 T_1 \frac{PM_{H_2O}}{\rho_{H_2O}} + \Delta H_v + Cp_{H_2O} PM_{H_2O} T_1\right)} \\
& [(F_{CaO} Cp_{CaO} PM_{CaO} + F_{CaCO_3} Cp_{CaCO_3} PM_{CaCO_3})(T_{CaO} - \beta T_1) \\
& - \rho_1 Cp_1 T_1 (1-\beta)\left(F_{CaO}\frac{PM_{CaO}}{\rho_{CaO}} + F_{CaCO_3}\frac{PM_{CaCO_3}}{\rho_{CaCO_3}}\right) + Q_o \rho_o Cp_o T_o - q_1 \\
& + (R_1 \Delta H_s + (R_2 - R_3)\Delta H_c) V_1 - Q_o \rho_1 Cp_1 T_1]
\end{aligned} \tag{B.40}
$$

Application of exp.(B.40) to evaluate Ev_{H_2O} corresponds to the occurrence of boil-over. If $T_1 < T_b$, however, the temperature of the liquid is below boiling point. In this case, Ev_{H_2O} can be evaluated by

$$Ev_{H_2O} = A_t K_g (P^*_{H_2O} - P_{H_2O}) \tag{B.41}$$

where P_{H_2O} is the average water vapour pressure in the air volume of the vessel and $P^*_{H_2O}$ is the water vapour pressure at the surface of the liquid. For a limited temperature range, $P^*_{H_2O}$ can be approximated as a function of temperature (Perry and Chilton, 1973).

Heat Losses to the Surroundings

In a similar way, the heat losses to the atmosphere q can be evaluated assuming losses by convection only:

$$q = A_{lat} h (T - T_a) \tag{B.42}$$

Both h and K_g can be estimated for normal operating conditions (Welty, 1969).

Level Control

Proportional-only level controllers were included in the the slaker and causticizing tank models, according to:

$$Q_i = Q0_i + K(V_i - V0_i) \tag{B.43}$$

where $V0$ is the desired set-point for the volume of slurry and K is a proportionality constant.

Grit Losses

Grit losses were assumed to be a fraction of the solids feed, that is,

$$G = \beta(F_{CaO} + F_{CaCO_3}) \tag{B.44}$$

Liquor losses due to the removal of grit are not considered.

Estimation of Slurry Heat Capacity and Density

The heat capacity and density of the slurry in the causticizing tanks are considered to be equal, at any time, to those in the slaker. However, to compensate for changes in the green liquor concentration and solids feed rate, the heat capacity and density of the slurry in the slaker are recalculated at the beginning of each integration.

If m_{sol} is the mass of solids per unit volume of slurry,

$$m_{sol} = C_{CaO}PM_{Cao} + C_{Ca(OH)_2}PM_{Ca(OH)_2} + C_{CaCO_3}PM_{CaCO_3} \tag{B.45}$$

and V_{sol} is the volume of solids per unit volume of slurry, i.e.,

$$V_{sol} = \frac{C_{CaO}PM_{CaO}}{\rho_{CaO}} + \frac{C_{Ca(OH)_2}PM_{Ca(OH)_2}}{\rho_{Ca(OH)_2}} + \frac{C_{CaCO_3}PM_{CaCO_3}}{\rho_{CaCO_3}} \tag{B.46}$$

then

$$\rho_{susp} = m_{sol} + (1 - V_{sol})\rho_l \tag{B.47}$$

and

$$\begin{aligned} Cp_{susp} = {} & \frac{1}{m_{sol} + (1 - V_{sol})\rho_l}(C_{CaO}PM_{CaO}Cp_{Cao} + \\ & + C_{Ca(OH)_2}PM_{Ca(OH)_2}Cp_{Ca(OH)_2} + C_{CaCO_3}PM_{CaCO_3}Cp_{CaCO_3} + \\ & + (1 - V_{sol})\rho_l Cp_l) \end{aligned} \tag{B.48}$$

In the slaker, $\rho_o = \rho_l$, $\rho_1 = \rho_{susp}$, $Cp_o = Cp_l$ and $Cp_1 = Cp_{susp}$. Values for ρ_l as a function of concentration and temperature can be obtained from Kojo (1979b). Cp_l is considered to be constant.

The dynamic response of the tanks to start-up conditions and several disturbances is presented elsewhere (Pais and Portugal, 1994c).

Notation

A	=	Area (m^2)
C	=	Concentration (mol/m^3, unless otherwise specified)
Cp	=	Heat capacity (J/KgK)
Ev	=	Rate of evaporation of water (mol/s)
F	=	Molar flowrate (mol/s)
G	=	Flowrate of grit (mol/s)
h	=	Heat transfer coefficient (J/m^2sK)
$[i]$	=	Concentration of species i (gNa_2O/l)
k_1	=	Reaction rate constant for R_1 ($1/s$)
k_2'	=	Reaction rate constant for R_2 (l/gNa_2Omin)
k_3'	=	Reaction rate constant for R_3 ($(l/gNa_2O)^3/min$)
K	=	Constant in proportional level controller
K_g	=	Mass transfer coefficient for the gas phase (m/s)
m	=	Mass per unit volume (Kg/m^3)
P	=	Partial pressure (Pa)
PM	=	Molecular weight (Kg/mol)
q	=	Heat losses to the surroundings (J/s)
Q	=	Volumetric flowrate (m^3/s)
$Q0$	=	Set-point for volumetric flowrate (m^3/s)
R_1	=	Rate of slaking reaction (mol/m^3s)
R_2	=	Rate of forward causticizing reaction (mol/m^3s)
R_3	=	Rate of reverse causticizing reaction (mol/m^3s)
R	=	Universal gas constant ($J/molK$)
t	=	Time (s)
T	=	Temperature (K)
T_a	=	Ambient temperature (K)
V	=	Volume (m^3)
$V0$	=	Set-point for volume of reacting slurry (m^3)

Greek Letters

β	=	Fraction of solids feed to the slaker removed as grit
ΔH	=	Enthalpy of reaction or vaporization (J/mol)
ρ	=	Density (Kg/m^3)

Superscripts

in	=	Referred to initial conditions in the slaker ($t = 0$)
$*$	=	Referred to the gas phase in equilibrium with the liquid phase (at free surface of slurry)

Subscripts

b	=	Referred to boiling temperature
CaO	=	Referred to chemical species CaO
$Ca(OH)_2$	=	Referred to chemical species $Ca(OH)_2$
$CaCO_3$	=	Referred to chemical species $CaCO_3$
CO_3^{2-}	=	Referred to chemical species CO_3^{2-}
H_2O	=	Referred to chemical species H_2O
OH^-	=	Referred to chemical species OH^-
c	=	Referred to the causticizing reaction
l	=	Referred to the liquor

lat	=	Referred to the lateral area of the tank
o	=	Referred to inlet to slaker
s	=	Referred to the slaking reaction
sol	=	Referred to the total amount of solids present
susp	=	Referred to slurry
t	=	Referred to the surface of the slurry (on top of the tank)
v	=	Referred to vaporization of water
1	=	Referred to slaker (except R_1)

Appendix C

The White Liquor Clarifier

C.1 Introduction

Continuous clarification and thickening of solid-liquid slurries by gravity is an essential unit operation in many processes. In the chemical recovery cycle, the slurry from the causticizing units (lime mud) contains solids which must be removed from the white liquor before it is used for cooking the wood. The clarification of the white liquor is normally carried out in traditional gravity settlers (see Fig.C.1) Although design procedures for steady state operation are well

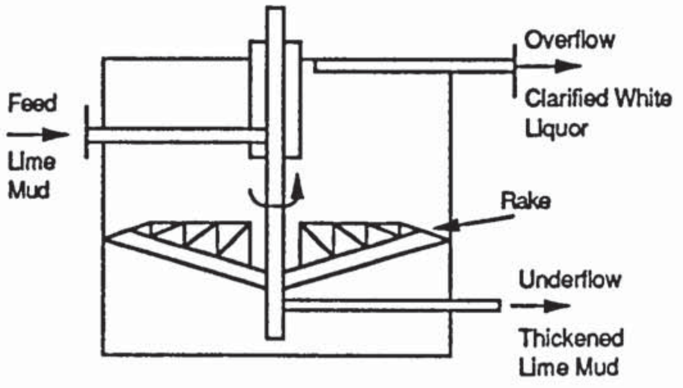


Figure C.1: Schematic representation of the conventional gravity settler.

established, the transient operation of the sedimentation process is not fully understood yet. Several attempts at dynamic simulation of continuous sedimentation have been reported (Alkema, 1971; Chi, 1974; Tracy and Keinath, 1974). All such simulation models are based on the known steady state behaviour of settling units, in which only discrete concentrations can exist, defined by the *limiting flux* for the system. Petty (1975) has obtained exact solutions to the equations that describe the transient state for certain feed and underflow changes, finding that the discrete-layer solution can include unallowable shock waves in the system. The correct solution may contain zones where the variation of the concentration with height is continuous; such solutions are known as rarefaction waves for analogous problems in gas dynamics. In some cases, the process dynamics differ markedly from prediction of models which admit unallowable discontinuities (Attir *et al.*, 1976). The possibility of the existence of zones where the variation of the concentration with height is continuous has therefore been accounted for in the model used in this work.

C.2 Formulation of Mathematical Model

The model used was based on the models proposed by Alkema (1971) and Chi (1974). It assumes the settler to be composed of four zones (bottom zone, Zone 1, Zone 2 and clarification zone) (see Fig.C.2). It is based on the following assumptions (Chi, 1974):

⋆ solids flux theory is valid;

⋆ the settler can be modelled as a series of completely mixed tanks;

Figure C.2: Zones considered in the model for the gravity settler. (From Chi, 1974).

⋆ the solids concentration is uniform within each layer;

⋆ no chemical reactions occur in the settler;

⋆ a higher concentration layer cannot exist above a lower concentration layer at any instant.

The second and third assumptions imply that there are finite discontinuities in the solids concentration at certain levels. As shown by Kynch (1952), in this case the differential equation of continuity no longer applies and must be replaced by a mass balance around the discontinuity (Shannon *et al.*, 1963). The propagation velocity of the discontinuity between concentrations C_i and C_{i+1}, δ_i, is given by:

$$\delta_i = \frac{G_i - G_{i+1}}{C_i - C_{i+1}} \quad i = 2, N + M \tag{C.1}$$

where the solids flux G is defined as

$$G = C(u + v) \tag{C.2}$$

and $u = \frac{q_u}{A}$.

The solids flux in the clarification zone is given by

$$G_c = C(u_o + v) \tag{C.3}$$

where $u_o = -\frac{q_o}{A}$

Since it has been assumed that the depth of the bottom zone and of the clarification zone remain constant, and that the feed well is at a fixed position, it follows that:

$$\delta_0 = \delta_1 = \delta_{N+M+1} = \delta_c = 0 \tag{C.4}$$

The depth of layer i, h_i, changes with rate

$$\frac{dh_i}{dt} = \delta_{i-1} - \delta_i \quad i = 1, N+M+1 \tag{C.5}$$

A mass balance around layer i yields

$$\frac{d}{dt}(h_i C_i) = (G_{i+1} - C_{i+1}\delta_i) - (G_i - C_i\delta_{i-1}) \tag{C.6}$$

Substitution of eq.(C.5) in eq.(C.6) gives

$$h_i \frac{dC_i}{dt} = (G_{i+1} - G_i) - \delta_i(C_{i+1} - C_i) \quad i = 1, N+M+1 \tag{C.7}$$

From eqs.(C.1), (C.4), (C.5) and (C.7) it follows that:

$$\frac{dC_1}{dt} = \frac{G_2 - G_1}{h_1} \tag{C.8}$$

$$h_i \frac{dC_i}{dt} = 0 \quad i = 2, N+M \tag{C.9}$$

and

$$\frac{dC_{N+M+1}}{dt} = \begin{cases} (G_{th} - G_{N+M+1})/h_{N+M+1} & M \neq 0 \\ 0 & M = 0 \end{cases} \tag{C.10}$$

where the solids flux into the thickening zones, G_{th}, is given by

$$G_{th} = \begin{cases} G_f + G_c & G_c > 0 \\ G_f & G_c \leq 0 \end{cases} \tag{C.11}$$

The equation that governs the concentration in the clarification zone is obtained making a mass balance:

$$\frac{dC_c}{dt} = \begin{cases} |G_o|/h_c & G_c > 0 \text{ and } G_o \neq 0 \\ (|G_o| - |G_c|)/h_c & \text{otherwise} \end{cases} \tag{C.12}$$

where

$$G_o = \begin{cases} -(G_{th} - G_{N+1}) & G_{N+1} < G_{th} \text{ and } M = 0 \\ 0 & \text{otherwise} \end{cases} \tag{C.13}$$

A flux into the clarification zone will take place only when an overloaded condition occurs, i.e., when the sludge blanket reaches the level of the feed well, thus $M = 0$, and the solids feed flux to the clarifier exceeds the maximum handling capacity of the settler, thus $G_{th} > G_{N+1}$. The solids concentrations of all layers in Zone 2, except possibly the top layer, and in Zone 1, are time invariant. The depth of each layer will however change with a rate given by eq.(C.5). If Zone 2 exists, i.e. when $h_{N+M+1} \neq 0$ and $M \neq 0$, then the solids concentration of the top layer varies with time according to eq.(C.10).

During a transient run, both the solids concentration and flowrate of the feed can change, as well as the underflow rate. New layers may form and existing layers disappear. The conditions for the appearance and disappearance of layers are as follows:

⋆ When the underflow rate is changed, a new layer of infinitesimal depth having a solids concentration equal to the limiting concentration C_{min}, corresponding to the new underflow velocity, forms somewhere in Zone 1. This new layer will develop in depth in time if the solids flux coming into it from the layer above is greater than its own flux, the limiting flux G_{min}. When this occurs, N is increased by 1.

⋆ At any instant, if G_{th} is less than G_{N+M+1}, which may happen if the feed condition and/or the underflow rate are changed, a new top layer will form in Zone 2, thus $M = M + 1$. The concentration of this new layer is that concentration which yields G_{th} at the particular u value at this instant, i.e., C_{th}. The depth of this layer changes according to eq.(C.5). On the contrary, no top layer will form in Zone 2 if $G_{th} \geq G_{N+M+1}$. Instead, the concentration of the layer imediately below the feed well will change accordingly to eq.(C.10).

⋆ The condition under which a layer i will disappear is simply that $h_i(t) \leq 0$ or $C_{i-1} \leq C_i$. In the latter case, h_{i-1} is replaced by $(h_{i-1} + h_i)$ if $h_i > 0$.

The dynamic equation for continuous sedimentation has been solved for step changes in the feed flux and underflow rate by Petty (1975) using the method of characteristics. If a comparison is made between the results of the model proposed by Chi (1974) and the solution obtained by Petty is made, striking differences can be observed. The simulation models fail because they consider only discontinuous, layered solutions, and solutions of this type must sometimes be excluded because of a condition known as *Lax's generalized entropy condition.* A discontinuity between two adjacent layers is not allowed if either of the following two conditions is satisfied (Attir *et al.*, 1976):

$$\frac{dG_i}{dC_i} > \frac{G_{i+1}(C_{i+1}) - G_i(C_i)}{C_{i+1} - C_i} \tag{C.14}$$

$$\frac{dG_i}{dC_i} < \frac{G_i(C_i) - G_{i-1}(C_{i-1})}{C_i - C_{i-1}} \tag{C.15}$$

If condition (C.14) is violated, then the discontinuity between layers i and $i + 1$ is an unallowed shock. The exact solution requires continuous variation of concentration with height in this region (a rarefaction wave); for simulation purposes, a series of discrete layers which span the concentration range may be inserted. Similarly, if condition (C.15) is violated then the discontinuity between layers i and $i - 1$ is an unallowed shock. The layers adjacent to the clarification and compaction zones are always retained, even if the concentration difference from an adjacent layer is less than the preset tolerance.

The system of ODE's was solved using subroutine LSODI (Hindmarsh, 1980). Because the solution at certain depths can be either continuous or discrete, depending on the verification of eqs(C.14) and (C.15), and, moreover, layers may appear or disappear, the number of dependent variables is not fixed. The variables must be re-ordered and re-numbered when layers are inserted or deleted, which leads to the need to reset the numerical process.

C.3 Settling Velocity

A very important parameter in either steady or transient state calculations is the solids settling velocity (v). In dilute slurries, the settling velocity of a particle can be assumed to be independent of the presence of other particles, according to Stokes' Law:

$$v_0 = \frac{(\rho_s - \rho_l) g D_p^2}{18\mu} \tag{C.16}$$

The settling velocity of the lime mud has been recognized to depend on a large number of factors such as the causticizing time (Uronen *et al.*, 1976, Bryce, 1980), the causticizing temperature (Olsen and Direnga, 1941; Uronen *et al.*, 1976; Uronen and Aurasmaa, 1979; Bryce, 1980; Mehra *et al.*, 1985; Olsen and Direnga, 1941), the ratio between the flowrate of lime feed and green liquor to the slaker (Uronen and Aurasmaa, 1979; Bryce, 1980; Kojo, 1980), the green liquor concentration (Uronen *et al.*, 1976; Campbell, 1981), the mixing conditions during slaking and causticizing (Olsen and Direnga, 1941; Dorris and Allen, 1987; Theliander and Grén, 1987), the calcination time (Uronen *et al.*, 1976), the calcination temperature (Uronen *et al.*, 1976), the size of the calcium carbonate particles (Uronen *et al.*, 1976) and the exposure of lime to air before slaking (Olsen and Direnga, 1941; Uronen *et al.*, 1976). The effect of each factor may be more or less obvious, but they all contribute to modifications in the chemical or physical properties of the solids particles and therefore to different behaviours in sedimentation. The unhindered settling velocity depends therefore on so many factors that it must be individually and experimentally adjusted for each specific case.

For concentrated slurries (*hindered settling*), moreover, the settling velocity depends on the local concentration C, i.e., $v = v(C)$. A large number of correlations has been found in the literature that attempts to describe the functional form of the settling velocity.

It is accepted (Shannon *et al.*, 1963) that the sedimentation velocity for rigid spheres is given by Richardson and Zaki's (1954) equation:

$$\frac{v}{v_0} = \varepsilon^{4.65} \tag{C.17}$$

Scott (1968) has analysed several experimental results (Tory, 1961; Comings, 1940; Hassett, 1964–65) for the sedimentation of calcium carbonate slurries, finding important discrepancies between them. Scott concluded that calcium carbonate slurries cannot consist of individually dispersed calcium carbonate particles. If the particles are assumed to be aggregated into sedimentation units with properties similar to flocs, the discrepancies can be explained. Michaels and Bolger (1962) have shown that such units immobilize a relatively large volume of water thus reducing the apparent settling velocity. The degree of aggregation, which determines the settling velocity of the slurry, is unknown and must be experimentally determined.

Other authors determined the settling velocity dependency with concentration by adjusting the sedimentation velocity curve, usually determined by experimental batch tests, to an interpolation function. This approach was used by

Chi (1974) and Jacobi and Williams (1973a, b). Jacobi and Williams, based on the results presented by Kinzner (1968), determined v_0 by an empirical regression correlation and then formulated the dependency of the hindered settling velocity with concentration by adjusting a piecewise linear appproximation to experimental data (approach similar to Chi's).

Another approach uses the values of the estimated slurry density and viscosity instead of the values for the liquid in Stokes' Law expression (Foust *et al.*, 1980). For the specific case under study, the viscosity and density of the white liquor can be estimated using the correlations presented by Kojo (1979b, c). This was the approach used in this study.

The effective viscosity of the slurry can be estimated by

$$\frac{\mu_B}{\mu} = \frac{10^{1.82(1-\mathcal{X})}}{\mathcal{X}} \tag{C.18}$$

where $\mathcal{X}$ is the volume fraction of liquid in the slurry, i.e., $\mathcal{X} = 1 - \frac{C}{\rho_s}$.

Similarly, the density of the slurry can be calculated from the densities of the liquid and the solids, and the solids concentration:

$$\rho_B = C + \mathcal{X}\rho_l \tag{C.19}$$

This leads to the following expression for the dependency of the solids settling velocity with concentration:

$$v = \frac{(\rho_s - (C + \mathcal{X}\rho_l))gD_p^2\mathcal{X}}{18\mu 10^{1.82(1-\mathcal{X})}} \tag{C.20}$$

As referred to above, the values for the density and viscosity of the liquid were estimated using the correlations presented by Kojo (1979b, c), and are taken as the values for the incoming stream from the last causticizer.

Notation

A	=	Cross-sectional area (m^2)
C	=	Solids concentration (Kg/m^3 slurry)
D_p	=	Average particle diameter (m)
g	=	Gravity acceleration (m/s^2)
G	=	Solids flux (Kg/m^2s)
h	=	Height of layer (m)
M	=	Number of layers in Zone 2
N	=	Number of layers in Zone 1
q	=	Volumetric flowrate (m^3/s)
u	=	Underflow velocity (m/s)
v	=	Settling velocity of solids (m/s)

Greek Letters

δ	=	Propagation velocity of concentration discontinuity
ε	=	Effective porosity of slurry
μ	=	Viscosity of liquid (Ns/m^2)
ρ	=	Density (Kg/m^3)
$\mathcal{X}$	=	Volume fraction of liquid in slurry ($\mathcal{X} = 1 - \frac{C}{\rho_s}$)

Subscripts

B	=	Referred to slurry
c	=	Referred to clarification zone
f	=	Referred to feed
i	=	Referred to layer i
l	=	Referred to liquid
min	=	Referred to minimum flux for specified operating conditions
o	=	Referred to overflow
s	=	Referred to solids
th	=	Referred to thickening zone
u	=	Referred to underflow
0	=	Referred to Stokes velocity

Appendix D

The Lime Mud Filter

D.1 Introduction

Liquid filtration is a fundamental unit operation which involves the separation, removal and collection of a discrete phase of matter existing in suspension (Cheremisinoff and Azbel, 1983). In the chemical recovery cycle of kraft paper pulp mills, caustic soda and water removal are essential before the calcination of the lime mud (Jacobi and Williams, 1973a), and are normally performed in a vacuum rotary pre-coat filter. This type of filter, where solids are discharged from the surface of a rotating drum by means of a scraper, air blowback, or a belt (Uronen, 1985), is essentially continuous. Periodical stops are still however necessary to clean the filtering medium. Fig.D.1 presents a schematic representation of a pre-coat filter; for a detailed description of the filtration equipment used in the industry, see Orr (1977). A fundamental issue is whether the lime

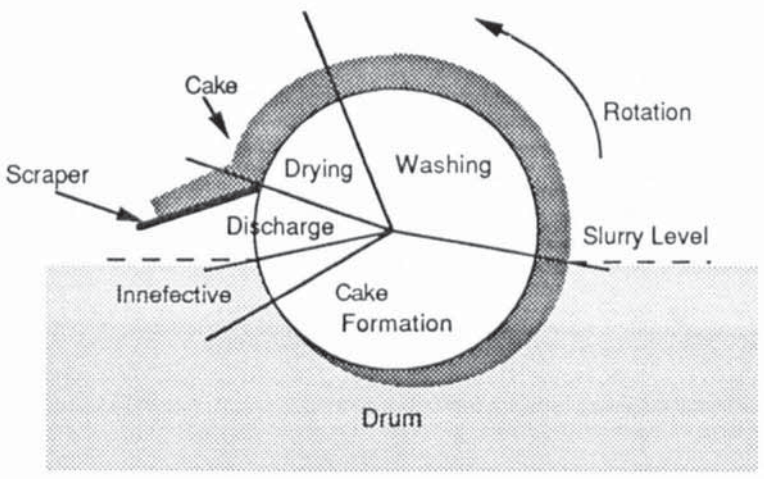


Figure D.1: Schematic representation of a rotary drum filter.

mud forms a compressible or a non-compressible cake. In the first case, if an accurate model is desired, then a set of partial differential equations as proposed by Stamatakis and Chen (1991) must be solved in order to determine the porosity profile across the cake. This profile can be used to, coupled to integration in time, determine the overall filtrate flowrate across the filter as the drum turns. If, on the other hand, the cake formed is incompressible, or if its compressibility is negligible, then a simplified model can be used in which both the porosity and the specific resistance of the cake are considered constant. Experimental data indicate that calcium carbonate slurries exhibit weak compressibility at low concentrations and appear to be incompressible for higher concentrations (Rushton and Katsoulas, 1984).

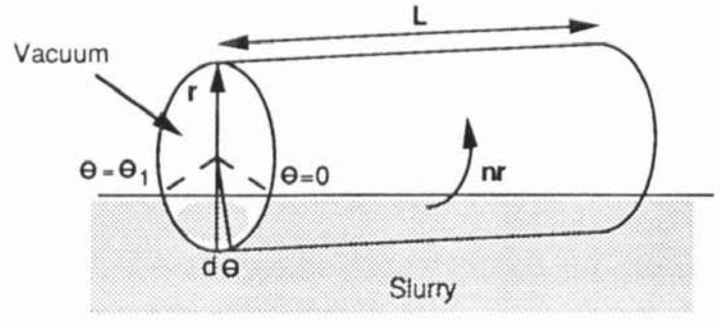


Figure D.2: Elementary area elements considered in the drum.

D.2 Mathematical Model

The following assumptions are considered in the model for filtration used in this work:

- ⋆ flow of filtrate through pores is laminar (large filter areas);
- ⋆ pressure drop is constant throughout filtration;
- ⋆ the amount of solids passing through the cake with the filtrate is negligible;
- ⋆ the resistance of the filter medium is negligible compared to the resistance of the cake.

The basic equation for filtration is (Azbel, 1983)

$$u = \frac{1}{A}\frac{dV}{dt} = \frac{\Delta P}{\mu(R_f + R_c)} \tag{D.1}$$

The differential area of the element located at position θ (see Fig.D.2) is $dA = rLd\theta$.

The amount of filtrate that passes, per second, through the cake element located at position θ is

$$udA = \frac{rL\Delta P}{\mu[R_f + R_c(\theta)]}d\theta \tag{D.2}$$

Through all the submerged area of the filter, per second, the amount of filtrate collected is

$$q_{filt} = \int_0^{\theta_1} \frac{rL\Delta P}{\mu[R_f + R_c(\theta)]}d\theta \tag{D.3}$$

It is now necessary to determine $R_c(\theta)$, since the resistance that the cake opposes to the filtration process depends on the height of cake formed so far. $R_c(\theta)$ can be expressed as $R_c(\theta) = r_o h_c$ where h_c is the height of cake formed. Furthermore, let x_o denote the ratio of cake volume to filtrate volume, i.e., $x_o = \frac{h_c A}{V}$; this implies that $\frac{dV}{dt} = \frac{A}{x_o}\frac{dh_c}{dt}$. If this expression for $\frac{dV}{dt}$ is used in eq.(D.1), then

$$\frac{dh_c}{dt} = \frac{x_o \Delta P}{\mu(R_f + r_o h_c)} \tag{D.4}$$

which upon integration between 0 and t, and between 0 and h_c ($\Delta P, x_o, \mu, R_f$ and r_o are assumed constant in time) yields

$$R_f h_c + r_o \frac{h_c^2}{2} = \frac{x_o \Delta P}{\mu} t \tag{D.5}$$

The relationship between t and θ is immediate ($t = \frac{\theta}{n_r}$) so eq.(D.5) is equivalent to

$$h_c = \frac{-R_f + \left(R_f^2 + \frac{2 r_o x_o \Delta P \theta}{n_r \mu}\right)^{\frac{1}{2}}}{r_o} \tag{D.6}$$

Eq.(D.6) represents the dependency of h_c with θ (or t) and can be used in eq.(D.3) yielding

$$q_{filt} = \frac{rL\Delta P}{\mu} \int_0^{\theta_1} \frac{d\theta \cdot}{\left(R_f^2 + \frac{2 r_o x_o \Delta P \theta}{\mu n_r}\right)^{\frac{1}{2}}} \tag{D.7}$$

If R_f is assumed to be negligible, then upon integration

$$q_{filt} = rL \left(\frac{2\Delta P n_r \theta_1}{r_o x_o \mu}\right)^{\frac{1}{2}} \tag{D.8}$$

It is sometimes preferable to use r_w and x_w, such that x_w is the mass of solids deposited in cake per unit volume of filtrate, i.e.,

$$x_w = \frac{c_{feed}}{1 - \frac{c_{feed}}{\rho_s} - \frac{\varepsilon c_{feed}}{(1-\varepsilon)\rho_s}} \tag{D.9}$$

In this case r_w corresponds to the specific mass cake resistance and is such that $r_w x_w = r_o x_o$. Eq.(D.8) becomes

$$\frac{dV}{dt} = rL \left(\frac{2\Delta P n_r \theta_1}{r_w x_w \mu}\right)^{\frac{1}{2}} \tag{D.10}$$

Furthermore, if the characteristics of the cake are known, it is possible to evaluate all the volumetric flowrates involved:

$$q_{filt} = \frac{dV}{dt} \tag{D.11}$$

$$q_{susp} = \frac{dV}{dt} \frac{1}{1 - \frac{c_{feed}}{\rho_s} - \frac{\varepsilon c_{feed}}{(1-\varepsilon)\rho_s}} \tag{D.12}$$

$$q_c = q_{susp} \left[\frac{c_{feed}}{\rho_s} + \frac{\varepsilon c_{feed}}{(1-\varepsilon)\rho_s}\right] \tag{D.13}$$

If drying is assumed to reduce the water in the cake by a fraction of f_{dry}, i.e., the amount of water present in the cake per unit volume of cake is $\varepsilon \rho_l f_{dry}$, then the mass fraction of solids in the lime mud that goes to the kiln is

$$f_{sol} = \frac{(1-\varepsilon)\rho_s}{(1-\varepsilon)\rho_s + \varepsilon \rho_l f_{dry}} \tag{D.14}$$

Mass flowrates, after drying, from the mud filter can be evaluated by simple mass balances. The model for the lime mud filter used in this work is a very simplified model, where the dynamics of the filter are neglected (i.e., the filter is supposed to be in steady state at all times) and factors such as periodical stops for cleaning, etc. are ignored. Also, the washing stage was not considered and the model for the drying stage is again a simplified one, and is simulated simply by reducing the amount of liquid left in the cake by a specified fraction.

Notation

A	=	Filtration area (m^2)
C	=	Concentration of solids in feed slurry (Kg/m^3)
f_{dry}	=	Fraction of water in cake remaining after the drying process
f_{sol}	=	Mass fraction of solids in the lime mud to the kiln
h	=	Height of cake formed (m)
L	=	Length of the drum (m)
n_r	=	Rotation speed of drum (rad/s)
P	=	Pressure (Pa)
q	=	Volumetric flowrate (m^3/s)
r	=	Radius of the drum (m) (except r_o and r_w)
r_o	=	Specific volumetric cake resistance (m^{-2})
r_w	=	Specific mass cake resistance (m/Kg)
R	=	Resistance to flow (m^{-1})
t	=	Time (s)
u	=	Linear velocity of filtrate through cake (m/s)
V	=	Volume of filtrate (m^3)
x_o	=	Ratio of cake volume to filtrate volume
x_w	=	Ratio of mass of solids in cake to filtrate volume (Kg/m^3)

Greek Letters

Δ	=	Difference (of pressure)
ε	=	Porosity of cake
μ	=	Viscosity (Ns/m^2)
ρ	=	Density (Kg/m^3)
θ	=	Angular coordinate of drum (rad)

Subscripts

c	=	Referred to cake
f	=	Referred to filter medium
$feed$	=	Referred to feed slurry
$filt$	=	Referred to filtrate
l	=	Referred to liquid
o	=	Referred to volume
s	=	Referred to solids
$susp$	=	Referred to suspension
w	=	Referred to mass
1	=	Referred to end of submersion

Appendix E

Software Developed

E.1 Installation of the Prototype

In order to install the software developed in this work, download, via anonymous ftp, the XProc.tar file from the cs.aston.ac.uk site, bg/write.me/fatima directory, and place it in your home directory. Extract the XProcSim directory and subdirectories with tar xvf XProc.tar; change your working directory to the XProcSim directory (cd XProcSim) and type make to compile. (A makefile is supplied in the XProcSim/object_files directory). The final executable will be placed in the XProcSim/bin directory under the name simul. The PVM software must also be available and properly installed; please see the PVM 3 User's Guide and Reference Manual (Geist *et al.*, 1994) for further instructions.

E.2 XProcSim Directory

```
total 28
drwx------  9 paisfic  cs_rsrch    1536 Jan 24 17:51 .
drwx------ 28 paisfic  cs_rsrch    1536 Jan 24 17:33 ..
drwx------  2 paisfic  cs_rsrch     512 Jan 24 17:47 bin
drwx------  2 paisfic  cs_rsrch     512 Jan 13 18:24 data_files
drwx------  2 paisfic  cs_rsrch    1536 Jan 13 18:32 icons
drwx------  8 paisfic  cs_rsrch     512 Jan 22 16:13 include
-rwx--x--x  1 paisfic  cs_rsrch      34 Jan 24 17:48 make
drwx------  2 paisfic  cs_rsrch    1024 Jan 24  1995 object_files
drwx------  2 paisfic  cs_rsrch    1536 Jan 24 13:57 result_files
drwx------  8 paisfic  cs_rsrch     512 Jan 24 17:51 src
```

E.3 XProcSim/bin Subdirectory

```
total 3526
drwx------  2 paisfic  cs_rsrch     512 Jan 17 16:46 .
drwx------ 13 paisfic  cs_rsrch    1536 Jan 18 13:44 ..
-rwx------  1 paisfic  cs_rsrch  255208 Jan 17 16:46 dbat_slave
-rwx------  1 paisfic  cs_rsrch  374416 Jan 17 16:46 dk_slave
-rwx------  1 paisfic  cs_rsrch  253852 Jan 17 16:46 dwlc_slave
-rwx------  1 paisfic  cs_rsrch  875540 Jan 17 16:46 simul
```

E.4 XProcSim/data_files Subdirectory

```
total 66
drwx------  2 paisfic  cs_rsrch     512 Jan 13 18:24 .
drwx------  9 paisfic  cs_rsrch    1536 Jan 24 17:51 ..
-rw-------  1 paisfic  cs_rsrch     676 Jan 13 16:02 bat_bar.d
-rw-------  1 paisfic  cs_rsrch     323 Jan 13 15:42 bat_line.d
-rw-------  1 paisfic  cs_rsrch     439 Nov 13 14:32 chemrec.d
-rw-------  1 paisfic  cs_rsrch    2250 Jul  7  1994 dbat.d
-rw-------  1 paisfic  cs_rsrch   14988 Jun 15  1994 dk.d
-rw-------  1 paisfic  cs_rsrch     444 Nov 14 12:51 dmf.d
-rw-------  1 paisfic  cs_rsrch     646 Nov 30 17:44 dwlc.d
-rw-------  1 paisfic  cs_rsrch     340 Jan  5 13:26 kiln_line.d
-rw-------  1 paisfic  cs_rsrch    4688 Jan  6 16:11 old_kiln_line.d
-rw-------  1 paisfic  cs_rsrch     284 Jun 16  1994 shape
```

E.5 XProcSim/icons Subdirectory

```
total 134
drwx------  2 paisfic  cs_rsrch    1536 Jan 13 18:32 .
drwx------ 13 paisfic  cs_rsrch    1536 Jan 18 13:44 ..
-rw-------  1 paisfic  cs_rsrch     222 Feb 16  1994 Cross_hair_cursor.icon
-rw-------  1 paisfic  cs_rsrch    1278 Feb 16  1994 Deletion.icon
-rw-------  1 paisfic  cs_rsrch     222 Feb 16  1994 Hand_cursor.icon
-rw-------  1 paisfic  cs_rsrch    1278 Feb 16  1994 Selection.icon
```

```
-rw-------   1 paisfic  cs_rsrch      222 Feb 16  1994 Selection_cursor.icon
-rw-------   1 paisfic  cs_rsrch      222 Feb 21  1994 Text_cursor.icon
-rw-------   1 paisfic  cs_rsrch     2192 Apr  6  1994 bar_chart.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 causticizer1_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 causticizer1_diagram_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 causticizer2_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 causticizer2_diagram_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 causticizer3_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 causticizer3_diagram_mask.icon
-rw-------   1 paisfic  cs_rsrch     2183 Apr  6  1994 data_panel.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 data_panel_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     2190 May  4  1994 draw.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 green_liquor_clarifier_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 green_liquor_clarifier_diagram_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 lime_kiln_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 lime_kiln_diagram_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 line_chart.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 long_flowsheet_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 main.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 main_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 mud_filter_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 mud_filter_diagram_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 slaker_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 slaker_diagram_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 wait.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 wait_mask.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 white_liquor_clarifier_diagram.icon
-rw-r--r--   1 paisfic  cs_rsrch     1997 Apr  6  1994 white_liquor_clarifier_diagram_mask.icon
```

E.6 XProcSim/include Subdirectory

```
total 26
drwx------   8 paisfic  cs_rsrch      512 Jan 22 16:13 .
drwx------   9 paisfic  cs_rsrch     1536 Jan 24 17:51 ..
-rw-------   1 paisfic  cs_rsrch      813 Jan 17 13:13 Xfiles
drwx------   2 paisfic  cs_rsrch      512 Jan 22 17:45 adaptors
drwx------   3 paisfic  cs_rsrch     1024 Jan 22 17:40 controllers
drwx------   2 paisfic  cs_rsrch      512 Jan 22 17:50 exception_handlers
drwx------   2 paisfic  cs_rsrch      512 Jan 22 17:47 graphical_items
drwx------   3 paisfic  cs_rsrch      512 Jan 23 17:08 models
-rw-------   1 paisfic  cs_rsrch      519 Nov 24 13:36 passthrough_procedures.h
-rw-------   1 paisfic  cs_rsrch     1423 Jan 22 17:26 simul.h
drwx------   3 paisfic  cs_rsrch     1024 Jan 22 17:30 views
```

E.7 XProcSim/include/adaptors Subdirectory

```
total 18
drwx------   2 paisfic  cs_rsrch      512 Jan 22 17:45 .
drwx------   8 paisfic  cs_rsrch      512 Jan 22 16:13 ..
-rw-------   1 paisfic  cs_rsrch       73 Nov 22 17:44 Adaptor.h
-rw-------   1 paisfic  cs_rsrch      233 Nov 14 12:35 Caust_Battery_Model_Bar_Chart_Adaptor.h
-rw-------   1 paisfic  cs_rsrch      236 Nov 14 12:36 Caust_Battery_Model_Line_Chart_Adaptor.h
-rw-------   1 paisfic  cs_rsrch      209 Nov 14 12:36 Kiln_Model_Line_Chart_Adaptor.h
-rw-------   1 paisfic  cs_rsrch      185 Nov 14 12:37 Model_Adaptor.h
-rw-------   1 paisfic  cs_rsrch      196 Jan 22 17:47 adaptor.h
-rw-------   1 paisfic  cs_rsrch      164 Sep 12 09:37 adaptor_class.h
```

E.8 XProcSim/include/controllers Subdirectory

```
total 48
drwx------   3 paisfic  cs_rsrch     1024 Jan 22 17:40 .
drwx------   8 paisfic  cs_rsrch      512 Jan 22 16:13 ..
-rw-------   1 paisfic  cs_rsrch      202 Jun 17  1994 Bar_Chart_Controller.h
-rw-------   1 paisfic  cs_rsrch      569 Nov 14 12:32 Controller.h
-rw-------   1 paisfic  cs_rsrch      396 Jun 17  1994 Draw_Controller.h
-rw-------   1 paisfic  cs_rsrch      205 Nov 14 12:58 Draw_Fonts_Controller.h
-rw-------   1 paisfic  cs_rsrch      218 Nov 14 12:59 Draw_Load_File_Controller.h
```

```
-rw-------   1 paisfic  cs_rsrch     211 Nov 14 13:00 Draw_Palette_Controller.h
-rw-------   1 paisfic  cs_rsrch     214 Nov 14 13:01 Draw_Save_File_Controller.h
-rw-------   1 paisfic  cs_rsrch     205 Nov 14 13:02 Fill_Style_Controller.h
-rw-------   1 paisfic  cs_rsrch     190 Nov 14 13:02 Fonts_Controller.h
-rw-------   1 paisfic  cs_rsrch     220 Nov 14 13:18 Integration_Data_Controller.h
-rw-------   1 paisfic  cs_rsrch     202 Jun 21  1994 Line_Chart_Controller.h
-rw-------   1 paisfic  cs_rsrch     205 Nov 14 13:05 Line_Style_Controller.h
-rw-------   1 paisfic  cs_rsrch     205 Nov 14 13:05 Line_Width_Controller.h
-rw-------   1 paisfic  cs_rsrch     202 Nov 14 13:06 Load_File_Controller.h
-rw-------   1 paisfic  cs_rsrch     196 Nov 14 13:08 Palette_Controller.h
-rw-------   1 paisfic  cs_rsrch     200 Nov 14 13:09 Save_File_Controller.h
-rw-------   1 paisfic  cs_rsrch     190 Nov 14 13:11 String_Controller.h
-rw-------   1 paisfic  cs_rsrch     190 Nov 14 13:11 Units_Controller.h
drwx------   2 paisfic  cs_rsrch     512 Jan 20 15:39 chemrec
-rw-------   1 paisfic  cs_rsrch    1145 Jan 22 17:45 controller.h
-rw-------   1 paisfic  cs_rsrch     912 Nov  8 17:08 controller_class.h
```

E.9 XProcSim/include/controllers/chemrec Subdirectory

```
total 22
drwx------   2 paisfic  cs_rsrch     512 Jan 20 15:39 .
drwx------   3 paisfic  cs_rsrch    1024 Jan 22 17:40 ..
-rw-------   1 paisfic  cs_rsrch     253 Nov 14 13:14 Caust_Battery_Bar_Chart_Controller.h
-rw-------   1 paisfic  cs_rsrch     244 Jun 21  1994 Caust_Battery_Dimensions_Controller.h
-rw-------   1 paisfic  cs_rsrch     295 Jun 17  1994 Caust_Battery_Physical_and_Chemical_Data_
Controller.h
-rw-------   1 paisfic  cs_rsrch     320 Nov  8 17:56 Green_Liquor_Storage_Data_Controller.h
-rw-------   1 paisfic  cs_rsrch     235 Jun 17  1994 Lime_Kiln_Dimensions_Controller.h
-rw-------   1 paisfic  cs_rsrch     246 Nov 10 13:30 Lime_Kiln_Line_Chart_Controller.h
-rw-------   1 paisfic  cs_rsrch     286 Nov 14 13:22 Lime_Kiln_Physical_and_Chemical_Data_
Controller.h
-rw-------   1 paisfic  cs_rsrch     229 Nov 14 13:23 Mud_Filter_Diagram_Controller.h
-rw-------   1 paisfic  cs_rsrch     265 Jun 17  1994 White_Liquor_Clarifier_Diagram_Controller.h
```

E.10 XProcSim/include/exception_handlers Subdirectory

```
total 14
drwx------   2 paisfic  cs_rsrch     512 Jan 22 17:50 .
drwx------   8 paisfic  cs_rsrch     512 Jan 22 16:13 ..
-rw-------   1 paisfic  cs_rsrch    1428 Nov 24 13:12 Exception_Handler.h
-rw-------   1 paisfic  cs_rsrch     224 Jan  5 15:48 Master_Exception_Handler.h
-rw-------   1 paisfic  cs_rsrch      70 Jan 22 17:51 exception_handler.h
-rw-------   1 paisfic  cs_rsrch      58 Nov 14 15:03 exception_handler_class.h
```

E.11 XProcSim/include/graphical_items Subdirectory

```
total 52
drwx------   2 paisfic  cs_rsrch     512 Jan 22 17:47 .
drwx------   8 paisfic  cs_rsrch     512 Jan 22 16:13 ..
-rw-------   1 paisfic  cs_rsrch    1878 Jan  9 13:04 Basic_Item.h
-rw-------   1 paisfic  cs_rsrch      90 Nov 14 13:32 Graphical_Item.h
-rw-------   1 paisfic  cs_rsrch     250 Nov 14 13:32 Grid_Item.h
-rw-------   1 paisfic  cs_rsrch    1379 Jan 23 09:55 Init.h
-rw-------   1 paisfic  cs_rsrch     203 Nov 14 14:09 Mode_Item.h
-rw-------   1 paisfic  cs_rsrch    2428 Dec  2 17:27 Shape_List.h
-rw-------   1 paisfic  cs_rsrch    1417 Dec  2 17:56 Stream_Item.h
-rw-------   1 paisfic  cs_rsrch     611 Jan  9 13:05 Text_Item.h
-rw-------   1 paisfic  cs_rsrch     268 Nov 14 14:39 Unit_CB_Item.h
-rw-------   1 paisfic  cs_rsrch     258 Nov 14 14:40 Unit_FS_Item.h
-rw-------   1 paisfic  cs_rsrch     266 Nov 14 14:41 Unit_GLS_Item.h
```

```
-rw-------   1 paisfic  cs_rsrch     633 Nov 24 16:10 Unit_Item.h
-rw-------   1 paisfic  cs_rsrch     264 Nov 14 14:51 Unit_LK_Item.h
-rw-------   1 paisfic  cs_rsrch     264 Nov 14 14:54 Unit_LS_Item.h
-rw-------   1 paisfic  cs_rsrch     264 Nov 14 14:56 Unit_MF_Item.h
-rw-------   1 paisfic  cs_rsrch     271 Nov 14 14:57 Unit_WLC_Item.h
-rw-------   1 paisfic  cs_rsrch    1655 Jan 22 17:50 graphical_item.h
-rw-------   1 paisfic  cs_rsrch     318 Jan 22 17:50 graphical_item_class.h
```

E.12 XProcSim/include/models Subdirectory

```
total 20
drwx------   3 paisfic  cs_rsrch     512 Jan 23 17:08 .
drwx------   8 paisfic  cs_rsrch     512 Jan 22 16:13 ..
-rw-------   1 paisfic  cs_rsrch    1922 Jan 21 15:53 Model.h
-rw-------   1 paisfic  cs_rsrch     216 Jan 21 15:53 Stream_Model.h
-rw-------   1 paisfic  cs_rsrch     797 Jan 20 15:51 Super_Model.h
-rw-------   1 paisfic  cs_rsrch     133 Jan 20 15:51 Unit_Model.h
drwx------   2 paisfic  cs_rsrch    1024 Jan 20 15:51 chemrec
-rw-------   1 paisfic  cs_rsrch     808 Jan 23 17:04 model.h
-rw-------   1 paisfic  cs_rsrch     591 Jan 20 15:51 model_class.h
```

E.13 XProcSim/include/models/chemrec Subdirectory

```
total 42
drwx------   2 paisfic  cs_rsrch    1024 Jan 20 15:51 .
drwx------   3 paisfic  cs_rsrch     512 Jan 23 17:08 ..
-rw-------   1 paisfic  cs_rsrch    2415 Nov 28 14:08 Caust_Battery_Model.h
-rw-------   1 paisfic  cs_rsrch     507 Nov 11 10:43 From_Caust_Battery_to_Wlc_Stream_Model.h
-rw-------   1 paisfic  cs_rsrch     468 Nov 14 15:15 From_Filter_to_Kiln_Stream_Model.h
-rw-------   1 paisfic  cs_rsrch     247 Nov 14 15:15 From_Fuel_Storage_to_Lime_Kiln_Stream_
Model.h
-rw-------   1 paisfic  cs_rsrch     370 Nov 11 11:30 From_Green_Liquor_Storage_to_Caust_Battery_
Stream_Model.h
-rw-------   1 paisfic  cs_rsrch     593 Dec  5 10:37 From_Kiln_to_Slaker_Stream_Model.h
-rw-------   1 paisfic  cs_rsrch     278 Nov 11 11:33 From_Limestone_Storage_to_Lime_Kiln_Stream_
Model.h
-rw-------   1 paisfic  cs_rsrch     434 Nov 11 11:35 From_Wlc_to_Filter_Stream_Model.h
-rw-------   1 paisfic  cs_rsrch     134 Nov 11 11:36 Fuel_Storage_Model.h
-rw-------   1 paisfic  cs_rsrch     447 Nov 11 11:40 Green_Liquor_Storage_Model.h
-rw-------   1 paisfic  cs_rsrch    2700 Nov 11 10:39 Lime_Kiln_Model.h
-rw-------   1 paisfic  cs_rsrch     164 Nov 14 15:22 Limestone_Storage_Model.h
-rw-------   1 paisfic  cs_rsrch     811 Nov 15 09:42 Mud_Filter_Model.h
-rw-------   1 paisfic  cs_rsrch    1893 Nov 28 15:09 Wlc_Model.h
```

E.14 XProcSim/include/views Subdirectory

```
total 72
drwx------   3 paisfic  cs_rsrch    1024 Jan 22 17:30 .
drwx------   8 paisfic  cs_rsrch     512 Jan 22 16:13 ..
-rw-------   1 paisfic  cs_rsrch    1205 Jan 13 12:44 Bar_Chart_View.h
-rw-------   1 paisfic  cs_rsrch     317 Jan 17 10:43 Canvas_View.h
-rw-------   1 paisfic  cs_rsrch     315 Jan 17 10:21 Canvas_and_Panel_View.h
-rw-------   1 paisfic  cs_rsrch     316 Nov 18 10:50 Canvas_and_Two_Panels_View.h
-rw-------   1 paisfic  cs_rsrch    1879 Jan  6 17:08 Chart_View.h
-rw-------   1 paisfic  cs_rsrch     207 Jan 17 10:00 Command_Frame_View.h
-rw-------   1 paisfic  cs_rsrch     258 Jan 17 10:44 Command_Panel_View.h
-rw-------   1 paisfic  cs_rsrch     625 Nov 18 10:53 Draw_View.h
-rw-------   1 paisfic  cs_rsrch     478 Nov 26 12:31 Fill_Style_View.h
-rw-------   1 paisfic  cs_rsrch     439 Nov 26 12:32 Fonts_View.h
-rw-------   1 paisfic  cs_rsrch     155 Jan 17 09:59 Frame_View.h
-rw-------   1 paisfic  cs_rsrch     490 Nov 18 11:10 Integration_Data_View.h
-rw-------   1 paisfic  cs_rsrch    1202 Nov 22 17:43 Line_Chart_View.h
-rw-------   1 paisfic  cs_rsrch     477 Nov 26 12:32 Line_Style_View.h
-rw-------   1 paisfic  cs_rsrch     452 Nov 26 12:35 Line_Width_View.h
-rw-------   1 paisfic  cs_rsrch     317 Nov 26 12:35 Load_File_View.h
```

```
-rw-------   1 paisfic  cs_rsrch     311 Nov 26 12:36 Palette_View.h
-rw-------   1 paisfic  cs_rsrch     255 Jan 17 10:44 Panel_View.h
-rw-------   1 paisfic  cs_rsrch     324 Nov 26 12:37 Save_File_View.h
-rw-------   1 paisfic  cs_rsrch     283 Nov 26 12:36 String_View.h
-rw-------   1 paisfic  cs_rsrch     114 Jan 17 10:02 Super_View.h
-rw-------   1 paisfic  cs_rsrch     587 Nov 26 12:37 Units_View.h
-rw-------   1 paisfic  cs_rsrch    2619 Dec  2 17:38 View.h
-rw-------   1 paisfic  cs_rsrch     293 Nov 18 11:06 Wait_View.h
drwx------   2 paisfic  cs_rsrch    1024 Jan 20 17:11 chemrec
-rw-------   1 paisfic  cs_rsrch    1304 Jan 22 17:52 view.h
-rw-------   1 paisfic  cs_rsrch    1028 Dec  1 13:33 view_class.h
```

E.15 XProcSim/include/views/chemrec Subdirectory

```
total 30
drwx------   2 paisfic  cs_rsrch    1024 Jan 20 17:11 .
drwx------   3 paisfic  cs_rsrch    1024 Jan 22 17:30 ..
-rw-------   1 paisfic  cs_rsrch     344 Nov 18 11:06 Caust_Battery_Bar_Chart_View.h
-rw-------   1 paisfic  cs_rsrch     735 Nov 18 11:07 Caust_Battery_Dimensions_View.h
-rw-------   1 paisfic  cs_rsrch     628 Nov 18 11:08 Caust_Battery_Physical_and_Chemical_Data_
View.h
-rw-------   1 paisfic  cs_rsrch     338 Nov 18 11:09 Causticizer_Diagram_View.h
-rw-------   1 paisfic  cs_rsrch     787 Nov 18 11:09 Diagram_View.h
-rw-------   1 paisfic  cs_rsrch     478 Nov 18 11:21 Green_Liquor_Storage_Data_View.h
-rw-------   1 paisfic  cs_rsrch     346 Nov 18 11:10 Lime_Kiln_Diagram_View.h
-rw-------   1 paisfic  cs_rsrch     544 Nov 18 11:11 Lime_Kiln_Dimensions_View.h
-rw-------   1 paisfic  cs_rsrch     336 Nov 18 11:11 Lime_Kiln_Line_Chart_View.h
-rw-------   1 paisfic  cs_rsrch     861 Nov 18 11:12 Lime_Kiln_Physical_and_Chemical_Data_View.h
-rw-------   1 paisfic  cs_rsrch     382 Nov 18 11:12 Mud_Filter_Diagram_View.h
-rw-------   1 paisfic  cs_rsrch     324 Nov 18 11:13 Slaker_Diagram_View.h
-rw-------   1 paisfic  cs_rsrch     417 Nov 18 11:13 White_Liquor_Clarifier_Diagram_View.h
```

E.16 XProcSim/object_files Subdirectory

```
total 9642
drwx------   2 paisfic  cs_rsrch    1024 Jan 24 18:13 .
drwx------   9 paisfic  cs_rsrch    1536 Jan 24 17:51 ..
-rw-------   1 paisfic  cs_rsrch  342800 Jan 24 17:55 adaptor_meth.o
-rw-------   1 paisfic  cs_rsrch  474576 Jan 24 17:56 controller_meth.o
-rw-------   1 paisfic  cs_rsrch   51160 Jan 24 18:12 dbat.o
-rw-------   1 paisfic  cs_rsrch   15932 Jan 24 18:10 dbat_init.o
-rw-------   1 paisfic  cs_rsrch  188336 Jan 24 18:12 dbat_slave_wrapper.o
-rw-------   1 paisfic  cs_rsrch    6500 Jan 24 18:10 dbat_to_dwlc.o
-rw-------   1 paisfic  cs_rsrch    9736 Jan 24 18:10 dbat_writer.o
-rw-------   1 paisfic  cs_rsrch  224244 Jan 24 18:11 dk.o
-rw-------   1 paisfic  cs_rsrch   17444 Jan 24 18:10 dk_init.o
-rw-------   1 paisfic  cs_rsrch  190480 Jan 24 18:11 dk_slave_wrapper.o
-rw-------   1 paisfic  cs_rsrch    3432 Jan 24 18:10 dk_to_dbat.o
-rw-------   1 paisfic  cs_rsrch   11768 Jan 24 18:10 dk_writer.o
-rw-------   1 paisfic  cs_rsrch    5620 Jan 24 18:10 dmf.o
-rw-------   1 paisfic  cs_rsrch    6508 Jan 24 18:10 dmf_init.o
-rw-------   1 paisfic  cs_rsrch   41136 Jan 24 18:13 dwlc.o
-rw-------   1 paisfic  cs_rsrch    9516 Jan 24 18:10 dwlc_init.o
-rw-------   1 paisfic  cs_rsrch  187040 Jan 24 18:13 dwlc_slave_wrapper.o
-rw-------   1 paisfic  cs_rsrch    2744 Jan 24 18:10 dwlc_to_dmf.o
-rw-------   1 paisfic  cs_rsrch    9544 Jan 24 18:10 dwlc_writer.o
-rw-------   1 paisfic  cs_rsrch  334860 Jan 24 18:10 exception_handler_meth.o
-rw-------   1 paisfic  cs_rsrch  504664 Jan 24 17:57 graphical_item_meth.o
-rw-------   1 paisfic  cs_rsrch    4548 Jan 24 18:10 init_dk_to_dbat.o
-rw-------   1 paisfic  cs_rsrch  107148 Jan 24 14:15 ls.o
-rw-------   1 paisfic  cs_rsrch   13255 Jan 24 18:01 makefile
-rw-------   1 paisfic  cs_rsrch    8624 Jan 24 18:10 measure.o
-rw-------   1 paisfic  cs_rsrch  483064 Jan 24 17:55 model_meth.o
-rw-------   1 paisfic  cs_rsrch  328356 Jan 24 17:54 passthrough_procedures.o
-rw-------   1 paisfic  cs_rsrch  336752 Jan 24 17:52 simul.o
-rw-------   1 paisfic  cs_rsrch  843508 Jan 24 18:09 view_meth.o
```

E.17 XProcSim/result_files Subdirectory

```
total 36
drwx------  2 paisfic  cs_rsrch    2048 Jan 18 13:54 .
drwx------ 13 paisfic  cs_rsrch    1536 Jan 18 13:44 ..
-rw-------  1 paisfic  cs_rsrch     827 Jan 18 13:53 dbat.r0
-rw-------  1 paisfic  cs_rsrch   12199 Jan 18 13:53 dk.r0
-rw-------  1 paisfic  cs_rsrch     313 Jan 18 13:53 dwlc.r0
```

E.18 XProcSim/src Subdirectory

```
total 38
drwx------  8 paisfic  cs_rsrch     512 Jan 24 17:51 .
drwx------  9 paisfic  cs_rsrch    1536 Jan 24 17:51 ..
drwx------  2 paisfic  cs_rsrch     512 Jan 24 17:41 adaptors
drwx------  3 paisfic  cs_rsrch    1024 Jan 24 17:42 controllers
drwx------  2 paisfic  cs_rsrch     512 Jan 24 17:42 exception_handlers
drwx------  2 paisfic  cs_rsrch     512 Jan 24 17:42 graphical_items
drwx------  3 paisfic  cs_rsrch     512 Jan 24 17:42 models
-rw-------  1 paisfic  cs_rsrch    2773 Jan 24 17:53 passthrough_procedures.C
-rw-------  1 paisfic  cs_rsrch    5941 Jan 22 18:48 simul.C
drwx------  3 paisfic  cs_rsrch    2048 Jan 24 17:41 views
```

E.19 XProcSim/src/adaptors Subdirectory

```
total 24
drwx------  2 paisfic  cs_rsrch     512 Jan 24 17:41 .
drwx------  8 paisfic  cs_rsrch     512 Jan 24 17:51 ..
-rw-------. 1 paisfic  cs_rsrch      62 Jan 23 17:03 Adaptor.C
-rw-------  1 paisfic  cs_rsrch     946 Jan 23 17:03 Caust_Battery_Model_Bar_Chart_Adaptor.C
-rw-------  1 paisfic  cs_rsrch    2054 Jan 23 17:03 Caust_Battery_Model_Line_Chart_Adaptor.C
-rw----}---  1 paisfic  cs_rsrch    2870 Jan 23 17:03 Kiln_Model_Line_Chart_Adaptor.C
-rw-------  1 paisfic  cs_rsrch     150 Jan 23 17:03 Model_Adaptor.C
-rw-------  1 paisfic  cs_rsrch     214 Jan 23 17:14 adaptor_meth.C
```

E.20 XProcSim/src/controllers Subdirectory

```
total 100
drwx------  3 paisfic  cs_rsrch    1024 Jan 24 17:42 .
drwx------  8 paisfic  cs_rsrch     512 Jan 24 17:51 ..
-rw-------  1 paisfic  cs_rsrch    1721 Jan 23 16:58 Bar_Chart_Controller.C
-rw-------  1 paisfic  cs_rsrch     667 Jan 23 16:58 Controller.C
-rw-------  1 paisfic  cs_rsrch   15743 Jan 23 16:58 Draw_Controller.C
-rw-------  1 paisfic  cs_rsrch    1094 Jan 23 16:58 Draw_Fonts_Controller.C
-rw-------  1 paisfic  cs_rsrch    1445 Jan 23 16:58 Draw_Load_File_Controller.C
-rw-------  1 paisfic  cs_rsrch     748 Jan 23 16:58 Draw_Palette_Controller.C
-rw-------  1 paisfic  cs_rsrch    1347 Jan 23 16:58 Draw_Save_File_Controller.C
-rw-------  1 paisfic  cs_rsrch     544 Jan 23 16:58 Fill_Style_Controller.C
-rw-------  1 paisfic  cs_rsrch     716 Jan 23 16:58 Fonts_Controller.C
-rw-------  1 paisfic  cs_rsrch    2185 Jan 23 16:58 Integration_Data_Controller.C
-rw-------  1 paisfic  cs_rsrch    2056 Jan 23 16:59 Line_Chart_Controller.C
-rw-------  1 paisfic  cs_rsrch     569 Jan 23 16:59 Line_Style_Controller.C
-rw-------  1 paisfic  cs_rsrch     475 Jan 23 16:59 Line_Width_Controller.C
-rw-------  1 paisfic  cs_rsrch    1349 Jan 23 16:59 Load_File_Controller.C
-rw-------  1 paisfic  cs_rsrch     397 Jan 23 16:59 Palette_Controller.C
-rw-------  1 paisfic  cs_rsrch    1298 Jan 23 16:59 Save_File_Controller.C
-rw-------  1 paisfic  cs_rsrch     500 Jan 23 16:59 String_Controller.C
-rw-------  1 paisfic  cs_rsrch    1747 Jan 23 16:59 Units_Controller.C
drwx------  2 paisfic  cs_rsrch    1536 Jan 21 15:00 chemrec
-rw-------  1 paisfic  cs_rsrch    1164 Jan 23 17:13 controller_meth.C
```

E.21 XProcSim/src/controllers/chemrec Subdirectory

```
total 50
drwx------   2 paisfic  cs_rsrch     1536 Jan 21 15:00 .
drwx------   3 paisfic  cs_rsrch     1024 Jan 24 17:42 ..
-rw-------   1 paisfic  cs_rsrch     2103 Jan 23 17:00 Caust_Battery_Bar_Chart_Controller.C
-rw-------   1 paisfic  cs_rsrch     1701 Jan 24 14:26 Caust_Battery_Dimensions_Controller.C
-rw-------   1 paisfic  cs_rsrch     2036 Jan 24 14:38 Caust_Battery_Physical_and_Chemical_Data_
Controller.C
-rw-------   1 paisfic  cs_rsrch     1644 Jan 24 14:38 Green_Liquor_Storage_Data_Controller.C
-rw-------   1 paisfic  cs_rsrch     1955 Jan 24 14:28 Lime_Kiln_Dimensions_Controller.C
-rw-------   1 paisfic  cs_rsrch     1475 Jan 23 17:00 Lime_Kiln_Line_Chart_Controller.C
-rw-------   1 paisfic  cs_rsrch     6232 Jan 24 15:07 Lime_Kiln_Physical_and_Chemical_Data_
Controller.C
-rw-------   1 paisfic  cs_rsrch      418 Jan 23 17:00 Mud_Filter_Diagram_Controller.C
-rw-------   1 paisfic  cs_rsrch      488 Jan 23 17:00 White_Liquor_Clarifier_Diagram_Controller.C
```

E.22 XProcSim/src/exception_handlers Subdirectory

```
total 14
drwx------   2 paisfic  cs_rsrch      512 Jan 24 17:42 .
drwx------   8 paisfic  cs_rsrch      512 Jan 24 17:51 ..
-rw-------   1 paisfic  cs_rsrch     1697 Jan 23 17:03 Exception_Handler.C
-rw-------   1 paisfic  cs_rsrch     1114 Jan 23 17:03 Master_Exception_Handler.C
-rw-------   1 paisfic  cs_rsrch       89 Jan 23 17:14 exception_handler_meth.C
```

E.23 XProcSim/src/graphical_items Subdirectory

```
total 168
drwx------   2 paisfic  cs_rsrch      512 Jan 24 17:42 .
drwx------   8 paisfic  cs_rsrch      512 Jan 24 17:51 ..
-rw-------   1 paisfic  cs_rsrch     3480 Jan 23 17:01 Basic_Item.C
-rw-------   1 paisfic  cs_rsrch       87 Jan 23 17:01 Graphical_Item.C
-rw-------   1 paisfic  cs_rsrch     1090 Jan 23 17:01 Grid_Item.C
-rw-------   1 paisfic  cs_rsrch     6315 Jan 23 17:01 Init.C
-rw-------   1 paisfic  cs_rsrch     1153 Jan 23 17:01 Mode_Item.C
-rw-------   1 paisfic  cs_rsrch     7268 Jan 23 17:01 Shape_List.C
-rw-------   1 paisfic  cs_rsrch    17350 Jan 24 14:44 Stream_Item.C
-rw-------   1 paisfic  cs_rsrch     7427 Jan 23 17:01 Text_Item.C
-rw-------   1 paisfic  cs_rsrch     7451 Jan 23 17:02 Unit_CB_Item.C
-rw-------   1 paisfic  cs_rsrch     1377 Jan 23 17:02 Unit_FS_Item.C
-rw-------   1 paisfic  cs_rsrch     1538 Jan 23 17:02 Unit_GLS_Item.C
-rw-------   1 paisfic  cs_rsrch     9200 Jan 23 17:02 Unit_Item.C
-rw-------   1 paisfic  cs_rsrch     5027 Jan 23 17:02 Unit_LK_Item.C
-rw-------   1 paisfic  cs_rsrch     1382 Jan 23 17:02 Unit_LS_Item.C
-rw-------   1 paisfic  cs_rsrch     1376 Jan 23 17:02 Unit_MF_Item.C
-rw-------   1 paisfic  cs_rsrch     1423 Jan 23 17:02 Unit_WLC_Item.C
-rw-------   1 paisfic  cs_rsrch      417 Jan 23 17:14 graphical_item_meth.C
```

E.24 XProcSim/src/models Subdirectory

```
total 30
drwx------   3 paisfic  cs_rsrch      512 Jan 24 17:42 .
drwx------   8 paisfic  cs_rsrch      512 Jan 24 17:51 ..
-rw-------   1 paisfic  cs_rsrch     2045 Jan 23 16:51 Model.C
-rw-------   1 paisfic  cs_rsrch      211 Jan 23 16:51 Stream_Model.C
-rw-------   1 paisfic  cs_rsrch     5326 Jan 24 16:02 Super_Model.C
-rw-------   1 paisfic  cs_rsrch      134 Jan 23 16:52 Unit_Model.C
drwx------   2 paisfic  cs_rsrch     1536 Jan 21 15:06 chemrec
-rw-------   1 paisfic  cs_rsrch      827 Jan 23 17:13 model_meth.C
```

E.25 XProcSim/src/models/chemrec Subdirectory

```
total 428
drwx------   2 paisfic  cs_rsrch   1536 Jan 21 15:06 .
drwx------   3 paisfic  cs_rsrch    512 Jan 24 17:42 ..
-rw-------   1 paisfic  cs_rsrch  12080 Jan 23 16:52 Caust_Battery_Model.C
-rw-------   1 paisfic  cs_rsrch   2746 Jan 23 16:52 From_Caust_Battery_to_Wlc_Stream_Model.C
-rw-------   1 paisfic  cs_rsrch   2233 Jan 23 16:52 From_Filter_to_Kiln_Stream_Model.C
-rw-------   1 paisfic  cs_rsrch    286 Jan 23 16:52 From_Fuel_Storage_to_Lime_Kiln_Stream_
Model.C
-rw-------   1 paisfic  cs_rsrch    696 Jan 23 16:52 From_Green_Liquor_Storage_to_Caust_Battery_
Stream_Model.C
-rw-------   1 paisfic  cs_rsrch   3276 Jan 23 16:52 From_Kiln_to_Slaker_Stream_Model.C
-rw-------   1 paisfic  cs_rsrch    306 Jan 23 16:52 From_Limestone_Storage_to_Lime_Kiln_Stream_
Model.C
-rw-------   1 paisfic  cs_rsrch   1806 Jan 23 16:53 From_Wlc_to_Filter_Stream_Model.C
-rw-------   1 paisfic  cs_rsrch    136 Jan 23 16:53 Fuel_Storage_Model.C
-rw-------   1 paisfic  cs_rsrch   1074 Jan 23 16:53 Green_Liquor_Storage_Model.C
-rw-------   1 paisfic  cs_rsrch  12696 Jan 23 16:53 Lime_Kiln_Model.C
-rw-------   1 paisfic  cs_rsrch    161 Jan 23 16:53 Limestone_Storage_Model.C
-rw-------   1 paisfic  cs_rsrch   3133 Jan 23 16:53 Mud_Filter_Model.C
-rw-------   1 paisfic  cs_rsrch   8847 Jan 23 16:53 Wlc_Model.C
-rw-------   1 paisfic  cs_rsrch  18984 Dec  1 17:27 dbat.f
-rw-------   1 paisfic  cs_rsrch   3074 Jan 12 11:03 dbat_init.f
-rw-------   1 paisfic  cs_rsrch   6928 Jan 23 17:32 dbat_slave_wrapper.C
-rw-------   1 paisfic  cs_rsrch   2088 Dec  5 16:09 dbat_to_dwlc.f
-rw-------   1 paisfic  cs_rsrch   1509 Jan 12 11:17 dbat_writer.f
-rw-------   1 paisfic  cs_rsrch  63188 Dec  1 15:44 dk.f
-rw-------   1 paisfic  cs_rsrch   3084 Jan 12 11:08 dk_init.f
-rw-------   1 paisfic  cs_rsrch   7460 Jan 23 17:32 dk_slave_wrapper.C
-rw-------   1 paisfic  cs_rsrch   1396 Dec  5 10:35 dk_to_dbat.f
-rw-------   1 paisfic  cs_rsrch   4570 Jan 12 11:18 dk_writer.f
-rw-------   1 paisfic  cs_rsrch   2146 Dec  1 09:55 dmf.f
-rw-------   1 paisfic  cs_rsrch    783 Jan 12 12:28 dmf_init.f
-rw-------   1 paisfic  cs_rsrch  18361 Dec  1 17:23 dwlc.f
-rw-------   1 paisfic  cs_rsrch   1361 Jan 12 11:11 dwlc_init.f
-rw-------   1 paisfic  cs_rsrch   5424 Jan 23 17:32 dwlc_slave_wrapper.C
-rw-------   1 paisfic  cs_rsrch    557 Jun 11  1994 dwlc_to_dmf.f
-rw-------   1 paisfic  cs_rsrch   1502 Jan 12 11:25 dwlc_writer.f
-rw-------   1 paisfic  cs_rsrch   1889 Dec  5 10:35 init_dk_to_dbat.f
-rw-------   1 paisfic  cs_rsrch   2318 Jan 10 18:03 measure.f
```

E.26 XProcSim/src/views Subdirectory

```
total 266
drwx------   3 paisfic  cs_rsrch   2048 Jan 24 17:41 .
drwx------   8 paisfic  cs_rsrch    512 Jan 24 17:51 ..
-rw-------   1 paisfic  cs_rsrch  30892 Jan 24 17:59 Bar_Chart_View.C
-rw-------   1 paisfic  cs_rsrch   1537 Jan 23 16:54 Canvas_View.C
-rw-------   1 paisfic  cs_rsrch    560 Jan 23 16:54 Canvas_and_Panel_View.C
-rw-------   1 paisfic  cs_rsrch    500 Jan 23 16:54 Canvas_and_Two_Panels_View.C
-rw-------   1 paisfic  cs_rsrch   9071 Jan 23 16:54 Chart_View.C
-rw-------   1 paisfic  cs_rsrch    590 Jan 23 16:54 Command_Frame_View.C
-rw-------   1 paisfic  cs_rsrch    334 Jan 23 16:54 Command_Panel_View.C
-rw-------   1 paisfic  cs_rsrch   8218 Jan 24 18:03 Draw_View.C
-rw-------   1 paisfic  cs_rsrch   4888 Jan 23 16:54 Fill_Style_View.C
-rw-------   1 paisfic  cs_rsrch   4179 Jan 23 16:54 Fonts_View.C
-rw-------   1 paisfic  cs_rsrch    560 Jan 23 16:54 Frame_View.C
-rw-------   1 paisfic  cs_rsrch   3062 Jan 24 18:03 Integration_Data_View.C
-rw-------   1 paisfic  cs_rsrch  27203 Jan 24 18:03 Line_Chart_View.C
-rw-------   1 paisfic  cs_rsrch   4699 Jan 23 16:55 Line_Style_View.C
-rw-------   1 paisfic  cs_rsrch   3835 Jan 24 15:51 Line_Width_View.C
-rw-------   1 paisfic  cs_rsrch    769 Jan 23 16:55 Load_File_View.C
-rw-------   1 paisfic  cs_rsrch   2170 Jan 23 16:55 Palette_View.C
-rw-------   1 paisfic  cs_rsrch    423 Jan 23 16:55 Panel_View.C
-rw-------   1 paisfic  cs_rsrch    778 Jan 23 16:55 Save_File_View.C
-rw-------   1 paisfic  cs_rsrch    629 Jan 23 16:55 String_View.C
-rw-------   1 paisfic  cs_rsrch    211 Jan 23 16:56 Super_View.C
```

```
-rw-------   1 paisfic  cs_rsrch     6467 Jan 23 16:56 Units_View.C
-rw-------   1 paisfic  cs_rsrch     4275 Jan 23 16:56 View.C
-rw-------   1 paisfic  cs_rsrch     1598 Jan 23 16:56 Wait_View.C
drwx------   2 paisfic  cs_rsrch     1024 Jan 24 18:07 chemrec
-rw-------   1 paisfic  cs_rsrch     1323 Jan 23 17:13 view_meth.C
```

E.27 XProcSim/src/views/chemrec Subdirectory

```
total 280
drwx------   2 paisfic  cs_rsrch     1024 Jan 24 18:07 .
drwx------   3 paisfic  cs_rsrch     2048 Jan 24 17:41 ..
-rw-------   1 paisfic  cs_rsrch     1353 Jan 23 16:56 Caust_Battery_Bar_Chart_View.C
-rw-------   1 paisfic  cs_rsrch     6903 Jan 24 18:04 Caust_Battery_Dimensions_View.C
-rw-------   1 paisfic  cs_rsrch     5408 Jan 24 18:04 Caust_Battery_Physical_and_Chemical_Data_
View.C
-rw-------   1 paisfic  cs_rsrch     9944 Jan 24 18:04 Causticizer_Diagram_View.C
-rw-------   1 paisfic  cs_rsrch     3648 Jan 23 16:56 Diagram_View.C
-rw-------   1 paisfic  cs_rsrch     3447 Jan 24 18:05 Green_Liquor_Storage_Data_View.C
-rw-------   1 paisfic  cs_rsrch    20412 Jan 24 18:00 Lime_Kiln_Diagram_View.C
-rw-------   1 paisfic  cs_rsrch     5590 Jan 24 18:05 Lime_Kiln_Dimensions_View.C
-rw-------   1 paisfic  cs_rsrch     1166 Jan 23 16:57 Lime_Kiln_Line_Chart_View.C
-rw-------   1 paisfic  cs_rsrch    13432 Jan 24 18:06 Lime_Kiln_Physical_and_Chemical_Data_
View.C
-rw-------   1 paisfic  cs_rsrch    28803 Jan 24 18:06 Mud_Filter_Diagram_View.C
-rw-------   1 paisfic  cs_rsrch    11045 Jan 24 18:05 Slaker_Diagram_View.C
-rw-------   1 paisfic  cs_rsrch    22147 Jan 24 18:06 White_Liquor_Clarifier_Diagram_View.C
```