National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Π-DFD Graphic Interface: DFD Graphical Analysis within ET++ Framework using CWB Tool

Alison S. Greig

A Major Report

in

The Department

of

Computer Science

Presented in Partial Fulfilment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

April 1996

© Alison S. Greig, 1996

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-10854-6

Abstract

Π-DFD Graphic Interface. DFD Graphical Analysis within ET++ Framework using CWB Tool

Alison S. Greig

Π-DFD Graphic Interface is an object-oriented application implemented in C++ for a windowing environment. This system *displays* a graphical representation of a Data Flow Diagram and provides an interface for the *analysis* and simulation of the behavior of a Data Flow Diagram, it uses ET++ application framework for the display of the DFD, and Edinburgh Concurrency WorkBench (CWB) tool for the analysis of the DFD

It uses a formal description of a Data Flow Diagram to analyze and simulate the behavior of a DFD and provides a graphical tool for understanding the DFD. This is done by using the formalism provided with CCS to describe the Data Flow Diagram and CWB tool

It also provides a study of the applicability of the representing the DFD as a document so that a document application framework can be used. ET++ application framework was choosen since it provides all the necessary tools to display the DFD in a consistent way with other ET++ applications within a windowing environment. It provides by default all the scrolling functionality necessary when displaying a document.

What we have done can easily be expanded to any system that is a finite state machine that can be defined in CWB code. The challenge would be in defining the finite state machine in tuple format and modifying the display format of the entities.

// Acknowledgements

G. Butler was of great assistance throughout the project and especially with ET++ application framework. I. Tjandra helped enormously with his knowledge of CCS and CWB tool.

R. Shinghal and P. Grogono helped with their participation at meetings on the project with suggestions for the design and experience with decision making in software engineering projects.

Table of Contents

LIST OF FIGURES XI

1 INTRODUCTION 1

1.1 ANALYSIS OF DFD 2

1.1.1 CWB TOOL OVERVIEW 3

1.1.2 DFD REPRESENTED WITH CWB CODE 3

1.2 DESIGN GOALS 5

1.3 GRAPHICAL DISPLAY OF DFD 6

1.3.1 YOURDON STANDARD FOR DFD DISPLAY 7

1.3.2 ET++: DOCUMENT APPLICATION FRAMEWORK 7

1.4 CONTRIBUTIONS: CWB INTERFACE 10

1.4.1 ROLE OF CWB PROCESS 10

1.4.2 CWB DESIGN ISSUES 10

1.4.3 OUTPUT FOR CWB IN NORMAL MODE 11

1.4.4 STATUS OUTPUT FOR CWB IN SIMULATION MODE 11

1.4.5 ADDING NEW CWB COMMANDS 12

2 SPECIFICATION FOR Π-DFD GRAPHIC INTERFACE 13

2.1 PURPOSE 13

2.2 DEVELOPMENT ENVIRONMENT: ET++3.0 14

2.3 FACILITIES PROVIDED BY ET++ FOR ITS USER APPLICATIONS 14

2.4 THE Π-DFD GRAPHIC INTERFACE....15

2.4.1 MENUS PROVIDED....15

2.4.2 FUNCTIONALITY PROVIDED....15

2.5 FILE FACILITIES....16

2.5.1 FILE NAMING CONVENTION....16

2.5.2 FILE/NEW....18

2.5.3 FILE/OPEN....18

2.5.4 FILE/LOAD....18

2.5.4.1 DFD File already loaded....19

2.5.4.2 CWB Command Interpretation....19

2.5.5 FILE/CLOSE....19

2.5.6 FILE/SAVE....19

2.5.7 FILE/SAVEAS....19

2.5.8 FILE/PRINT....20

2.5.9 FILE/QUIT....20

2.5.9.1 CWB Command Input....20

2.6 STATE CHECKING FACILITIES....20

2.6.1 STATES/SIZE20

2.6.1.1 CWB Command Interpretation....20

2.6.1.2 CWB output format....21

2.6.2 STATES/DEADLOCK STATE....21

2.6.2.1 Deadlock Input....21

2.6.2.2 Deadlock Output....21

2.6.2.3 CWB Command Interpretation....22

2.7 EQUIVALENCE CHECKING **23**

2.7.1 EQUIVALENCE/CHECK DFD PROCESSES EQUIVALENCE 23

2.7.1.1 Input 23

2.7.1.2 Output 23

2.7.1.3 CWB command Interpretation 24

2.8 SIMULATION **24**

2.8.1 SIMULATION CONTROL 24

2.8.2 CWB COMMAND INTERPRETATION FOR SIMULATION 25

2.8.2.1 CWB input 25

2.8.2.2 CWB Output 25

2.8.2.3 Interpreting Potential Action 26

2.8.2.4 Displaying Potential Actions 27

2.8.2.5 Interpreting Process Status 27

2.8.2.6 Single Step through a DFD 28

2.8.3 INTERPRETATION OF CWB OUTPUT 29

2.8.3.1 CWB command output interpretation 29

2.8.4 SIMULATION/NB STEPS FOR RANDOM TRANSITION 30

2.8.4.1 CWB command interpretation 31

2.8.4 2 CWB Command Output 31

2.8.5 SIMULATION/SELECT PREVIOUS 32

2.8.5.1 CWB command input 32

2.8.5.2 CWB Command Output 32

2.8.6 SIMULATION/QUIT SIMULATION 33

2.9 GRAPHIC DISPLAY CONVENTIONS **33**

2.9.1 PROCESS DISPLAY STATUS 34

2.9.2 ACTION DISPLAY STATUS 34

2.9.3 GRAPHIC DISPLAY CONVENTIONS 35

2.9.3.1 Processes 35

2.9.3.2 Actions 35

2.10 FUTURE ENHANCEMENTS 36

2.10.1 MULTIPLE LEVELS OF DECOMPOSITION 36

2.10.1.1 Input 36

2.10.1.2 Output 37

2.10.1.3 Error 37

2.10.1.4 Closing a refinement 37

2.10.1.5 Further considerations 37

2.10.1.6 Display of hierarchical levels using multiple windows 38

2.10.2 SIMULATE A SUBSET OF DFD 38

2.10.3 DERIVATION COMMANDS 38

2.10.4 COMPARISON BETWEEN TWO OR MORE DATA FLOW DIAGRAMS 39

2.10.5 ON-SCREEN DIAGRAM EDITING 39

2.10.7 DIFFERENT DFD STANDARDS 40

2.11 THE LOOK OF THE INTERFACE 40

3 DESIGN 43

3.1 PRINCIPAL DESIGN DIAGRAMS 43

3.1.1 GLOBAL EVENT DIAGRAMS 43

3.1.2 ARCHITECTURAL DIAGRAMS 47

3.1.3 CLASS DIAGRAMS FOR DFD APPLICATION USER INTERFACE 50

3.1.4 CLASS DIAGRAMS FOR CWB INTERFACE ... 53

3.1.4.1 Communication with Cwb Process ... 53

3.1.4.2 Status of the Dfd ... 54

3.1.4.3 Interface between Application and CWB ... 54

3.1.4.4 Commands that interaction with Cwb Process ... 57

3.1.4.5 Command Events ... 59

3.2 DESIGN: USER INTERFACE ... 63

3.2.1 DFDDOCUMENT CLASS ... 63

3.2.2 DFDVIEW CLASS ... 67

3.2.3 CLASSES FOR STORAGE AND PRESENTATION OF SHAPES IN THE VIEW ... 69

3.2.3.1 Shape Class ... 69

3.2.3.2 Aline Class ... 70

3.2.3.3 LnShape Class ... 71

3.2.3.4 DProcess Class ... 73

3.2.3.5 PrShape Class ... 74

3.2.4 DIAGRAM COMMAND CLASSES ... 75

3.2.4.1 DFDCommand Class ... 76

3.2.4.2 RANCommand Class ... 77

3.3 DESIGN: Cwb INTERFACE ... 79

3.3.1 CWB PROCESS ... 79

3.3.1.1 CwbComm Class ... 80

3.3.2 CWB COMMANDS ... 82

3.3.2.1 CWBCommand Class ... 82

3.3.2.2 OpenFile Class ... 85

3.3.2.3 Deadlock Class ... 86

3.3.2.4 Size Class ... 88

3.3.2.5 Equivalence Class ... 89

3.3.3 SimCommands ... 90

3.3.3.1 SimCommand Class ... 90

3.3.3.2 Start Class ... 91

3.3.3.3 Stop Class ... 92

3.3.3.4 Next Class ... 93

3.3.3.5 Random Class ... 94

3.3.3.6 Previous Class ... 95

3.3.4 Dfd Status Classes ... 96

3.3.4.1 CwbLine Class ... 97

3.3.4.2 ProcessStatus Class ... 98

3.3.4.3 DfdStatus Class ... 100

3.3.4.4 PotentialAction Class ... 101

3.3.5 Map between Application and Cwb ... 104

3.3.5.1 Dfd Map Class ... 104

4 Conclusions ... 111

4.1 Future Improvements ... 112

4.1.1 Provide Editing capabilities ... 113

4.1.2 Provide Undo/Redo capabilities ... 114

4.1.3 Improve User Interface for Deadlock States ... 115

4.1.4 Making better use of CWB for Equivalence ... 116

4.1.5 More flexibility: selection of Process Name ... 116

4.1.6 Save Simulation DFD ... 117

4.1.7 ALLOW DFD WITH MULTIPLE LEVELS 117

4.2 POWER OF ET++ 118

4.2.1 ET++ DOCUMENTATION 119

4.2.2 COOKBOOK APPROACH 119

4.3 SUMMARY 120

4.4 PERSONAL NOTES 121

5 REFERENCES 122

List of Figures

FIGURE 2.11.1: Π-DFD GRAPHIC INTERFACE -- IDLE DATA FLOW DIAGRAM....40

FIGURE 2.11.2: Π-DFD GRAPHIC INTERFACE -- SIMULATION DATA FLOW DIAGRAM....41

FIGURE 3.1.1.1: GLOBAL EVENT DIAGRAM (1)....44

FIGURE 3.1.1.2: GLOBAL EVENT DIAGRAM (1)....46

FIGURE 3.1.2.1: TOP LEVEL ARCHITECTURE....48

FIGURE 3.1.2.2: ARCHITECTURE -- SUBSYSTEMS....49

FIGURE 3.1.3.1: OVERVIEW CLASS DIAGRAM....50

FIGURE 3.1.3.2: CLASS FOR GENERATING AND DISPLAY OF DIAGRAM SHAPES....51

FIGURE 3.1.3.3: CLASSES FOR DIAGRAM COMMANDS....52

FIGURE 3.1.4.2.1: DATA FLOW DIAGRAM STATUS CLASSES GENERALIZATION....55

FIGURE 3.1.4.3.1: DFD MAP CLASS AGGREGATION....56

FIGURE 3.1.4.4.1: CWBCOMMAND CLASS GENERALIZATION....58

FIGURE 3.1.4.5.1: DFD EVENTS: USER INPUT....60

FIGURE 3.1.4.5.2: CWBCOMMAND EVENTS....61

FIGURE 3.1.4.5.3: SIMCOMMAND EVENTS....62

1 Introduction

The primary aim of the Π-DFD project is to develop a software tool for re-engineering data flow diagrams of legacy systems. Taking the paper format of a data flow diagram, it is possible to create a formalized representation of the logical structure of the data flow diagram based on CCS (Calculus of Communicating Systems) [BGST 95B]. This form can be used by the Edinburgh Concurrency Workbench (CWB) for analysis of the data flow. The CWB tool can be used to examine various aspects of internal states of the data flow diagram, compare and contrast the equivalence of two diagrams, and it can be used to run simulations on the diagram interactively.

The CWB tool by itself is very powerful although its output is in CCS and modal logic. This makes it difficult to monitor the behavior of a small data flow diagram even for someone familiar with CCS!

This work is built on the work done in [BGST 95B] where a paper format of DFD is recognized and the semantic content extracted for understanding the DFD This work builds a graphic interface for this semantic understanding of the DFD in a display format readily understood by anyone who is familiar with the data flow diagram conventions The system provides interactive analysis of the states and behavior of the DFD

This was a joint effort with Valerie Large. For the implementation portion of the project, her efforts concerned mainly the application user interface while mine mainly concerned the CWB interface.

This project encompasses two main areas of study:

- Analyze and display the dynamic behavior of a DFD by making use of the formal representation of a DFD in CCS.
- Realize a system representing a data flow diagram as a *document* containing graphical and text entities so that it is possible to use a document application framework for the implementation of the system.

Π-DFD Graphic Interface uses two tools to accomplish this:

- CWB tool is used to interpret the CCS form to analyze the states and behavior of the data flow diagram; and
- ET++ is a document application framework that provides the architecture and the necessary tools for displaying and viewing a graphical document.

This software project consisted of determining which functionality is required to analyze a DFD, what commands from CWB tool are most useful for this purpose, specifying the system with respect to the user interface and the CWB interface, and designing and implementing the system.

The remainder of this section describe the background analysis done to determine the theoretical aspects of the project, the design issues considered, and implementation issues.

1.1 Analysis of DFD

In order to analyze the states and behavior of a data flow diagram, it is necessary to represent it in a logical format that is valid and usable.

1.1.1 CWB Tool Overview

CWB is an automated tool originally designed for the manipulation and analysis of concurrent systems defined in CCS. It can be used for any system that can be defined in CCS such as a data flow diagram [BGST 95A]. It reads the definition of the concurrent system from a *.CWB file.

We have studied it and selected the commands from CWB that are most useful for the analysis of a data flow diagram. . These commands are described in detail in Section 2 in each of the sections with the title CWB Command Interpretation.

There are two modes of operation in CWB: normal and simulation. The system is implemented according to these modes of operation. This makes it possible to analyze the states of the DFD before simulating the behavior of the DFD.

1.1.2 DFD represented with CWB Code

The CWB code that is defined in *.CWB file defines the DFD in an idle state. This means that none of the processes or actions are active. In non-mathematical terms, it basically defines the possible input/output actions that can activate each process in the DFD, and how processes connect together.

For the definition of a DFD, there are a few elements from CCS that are necessary.

+ represents or

. represents sequencing

| represents composition

\ represents restriction

a action a received by agent (input action to process)

'a action a transmitted by agent(output action from process)

There is only one command that is necessary for the definition of a DFD in CWB code:

`bi` binds the process name to the CCS definition of the process

Here is an example of a simple DFD defined in CWB:

```
bi input.'a.T1
bi E0 a.'r.E0
bi E1 r.'s.E1 + r.'v.E1
bi E2 s.'u.E2
bi E3 v.'b.E3 + u.'b.E3
bi b.'output.T2
bi DFD (T1|E0|E1|E2|E3|T2)\{r,s,v,u}
```

The second command defines the process `E0` for a DFD, indicating that the process `E0` can receive `a` as an input action. After receiving `a` as an input action, `E0` has an active status. This active status for `E0` after receiving `a` as an input action is represented in CCS as `'r.E0`. In order for `E0` to go back to the idle status, it has to have `r` as its output action. The idle status for `E0` in CCS is `E0`.

The last command defines how all the process in the DFD are composed. It can be interpreted as `DFD` contains `T1, E0, E1, E2, E3, T2`. However, each process in the data flow diagram does not function independently. They are restricted by the actions

`r, s, v, u`. This means that if `E0` needs to send `r` as its output action, it has to send it to a process that can receive `r` as its input action, namely `E1`.

The DFD, at any time, can be in 1 of many possible states. Each state of the DFD is defined by a unique set of process and action status called the status for the DFD. For example after process `E0` has received `a` as input action the status for `E0` is active The status for DFD is defined in CCS as: `(T1|'r.E0|E1|E2|E3|T2)\{r,s,v,u}`

1.2 Design goals

In order to provide flexibility in the design for future enhancements, it is important to distinguish between the different representations of the data flow diagram:

display format	standard used to display the items in the DFD
display position	has to distinguish between the internal and external coordinates making it easier to display different views of the DFD (zoom, pageup)
semantics	the underlying conceptual properties using CCS (Calculus of Communicating Systems) to define the data flow diagram

Display of DFD

It is important to keep the design of the display of the DFD as separate as possible from the semantics of the DFD. The analysis of the behavior of the DFD results in a change of status for the processes and actions in the DFD The resulting status for each process and

action is evaluated separately from the display of the status. This gives flexibility to the design making it possible in the future to change the display standard independently from the semantics of the DFD.

The position that is currently being displayed (external coordinates) should also be independent of the coordinate of each individual item within the DFD (internal coordinates).

Semantics of DFD

A design goal is to keep to a minimum the number of objects dependent on the CWB tool. The DFDMap provides an interface between the rest of the application and the CWB tool. CWB is responsible for keeping track of what dynamic behavior can occur in the DFD, and is a very powerful tool capable of doing all the analysis of the DFD when defined in CCS.

1.3 Graphical Display of DFD

In order to present the data flow diagram in a format that is easily understood to the user, it is has to be presented graphically and follow a well known and widely used standard.

For the implementation of the system, a good graphical interface is required that meet the specification and design requirements.

1.3.1 Yourdon standard for DFD display

We investigated several standards that exist for the display of data flow diagrams. DeMarco's, Yourdon Structured Method, Sommerville's, Rumbaugh and Ghezzi.

The Yourdon standard for DFD display was chosen because it provides a good standard that is well known by the software designers who will use the system, and thus makes the Π-DFD Graphic Interface easy to use.

1.3.2 ET++: document application framework

ET++3.0 (Editor Toolkit) [B 95, WGM 88] is a framework for building interactive applications which edit and display documents that contain text or graphical entities. It is a portable and homogeneous object-oriented class library and application framework. The work done here is on the UNIX platform with X-Windows for the windowing environment.

It provides an architecture which describes an application in terms of the major abstract classes Application, Document, and View.

It provides all the user interface building blocks, graphical and text items and basic data structures.

ET++ provides the high level application framework components that determines the flow of control for the application

ET++ provides all the default behavior for creating the application and viewing a document, and it provides all the standard menu items and functionality associated with the menu items. This implements a standard user interface for all ET++ applications.

Development with ET++ therefore requires less application-specific design and code than is required with a library of graphical tools.

How ET++ fits our design goal

ET++ provides an ideal architecture that provides all the tools necessary for the development of the graphic interface:

- it provides all the graphical tools required for the visual display of a DFD by providing graphic and text entities;
- it provides a default behavior that includes the standard *look* of the window application and the functionality for all the standard menu items necessary for a document application;
- it automatically takes care of all the scrolling ability, page up and page down; this results in a significant reduction of the amount of code required; and
- it provides all the control for the events in the windowing environment.

ET++ also satisfies the requirements defined in Section 1.2:

- it separates the internal and external coordinates that define the DFD; and
- it provides the architecture that separates the visual display (View) from the semantic content (Document) of the DFD.

These concepts adapt well to the ET++ document model which differentiates between the model of the document and the view of the document. Just as important, it provides the architecture that separates the view from the document. The view is just the display of the data flow diagram, while the document defines the content or semantics of the data flow diagram.

Representing DFD as a document

In order to take advantage of ET++, we have to represent the data flow diagram as a document; the DFD is viewed as a collection of graphic and text entities. Each object in the DFD has an associated position within the document. The content of the document is modified through the simulation process. If effect, the simulation modifies the appearance (status) of the objects in the document. Once, the DFD is considered to be a document, all the document viewing capabilities are by default provided by ET++

The modification of the display of the DFD through the simulation process is part of the application-specific behavior provided by Π-DFD Graphic Interface.

ET++ Conventions

While working within an application framework, it is important to follow the standards prescribed for that applications created within the framework For this reason, the CWB Interface follows the ET++3.0 Naming Conventions and ET++3.0 Meta-Information Conventions as described in [B 95].

1.4 Contributions: CWB Interface

My contribution dealt mostly with the interface with the CWB tool from the specification, to the design, the implementation, and testing to ensure that it functions as required.

This includes communication with the CWB process, parsing the input and output from the CWB process in the form of sending and receiving commands to CWB, interpreting the CCS data from CWB and translating it into a format useful for DFD status display. My efforts also include the interface with the user for the normal mode of operation.

1.4.1 Role of CWB process

In the design of the CWB interface, the CWB tool is completely responsible for the analysis of the data flow diagram, and the CWB interface just improves the output from CWB by extracting the information required to display the DFD. The DFDMap keeps track of the *current* status in the CWB and is dependent on CWB tool for determine the next action or previous actions.

1.4.2 CWB Design Issues

It was necessary to determine which functionality was required from CWB tool, and which CWB command best suited the needs for the display of a data flow diagram.

The challenge was to determine how it was possible to extract all the useful information from the CWB in terms of DFD status while in the simulation mode, to make the interface as flexible as possible for future development so that the modifications would be minimal,

to attempt to make the design as consistent as possible so that modifications are simplified, to try to make use of the same code as much as possible, to provide a consistent and easy to use interface to the CWB Interface, and to provide an output format from the CWB Interface (DFDMap) that is independent of the CWB format.

1.4.3 Output for CWB in Normal Mode

There is a good structure for adding new commands either in simulation mode or normal mode. For normal mode of operation, the CWB output requires some massaging to the output by displaying the result in a message dialog or a select item dialog.

1.4.4 Status Output for CWB in Simulation Mode

It is the simulation mode that modifies the status of each processes. Using the definitions in Section 1.1.2, the CWB Interface translates the CCS output to process and action status information.

All information for the potential actions are extracted including the future status. This is done by using the same mechanism as defining the current status and is useful if it is desired to modify the mechanism for selecting the simulation potential action (such as highlighting the changes that would occur if the simulation action was selected).

The status information is created as a collection of objects One or more actions are associated with each process as input and output actions. It has been designed,

implemented and tested for multiple input/output to a process, with no length restriction on the process or action labels.

1.4.5 Adding new CWB commands

The design has made it easy to add additional CWB commands. For the normal mode of operation, a new command would inherit the CwbCommand, and in simulation mode, a new command would inherit the SimCommand.

2 Specification for Π-DFD Graphic Interface

2.1 Purpose

The primary aim of the Π-DFD project is to develop software tools for re-engineering dataflow diagrams of legacy systems. A dataflow (DFD) diagram can be expressed in a tuple representation which represents the diagram in a textual format. During the course of the project a *translator* program was developed which would translate the tuple format into the CCS form recognizable by the Edinburgh Concurrency Workbench (CWB) Using the workbench the diagram can then be analyzed.

Using CWB, it is possible to explore DFDs from legacy systems before re-engineering: the CWB tool can be used to examine various aspects of internal states of the dataflow diagram, compare and contrast diagrams, and it can be used to run simulations on the diagram interactively. Initially, there was a textual interface to CWB: it exposed the user to the CCS form of diagram representation, CWB's own notation for state representation and modal calculus, none of which are user friendly. A graphic interface would make the CWB facilities more accessible by shielding the user from CWB notation and by providing a visual interpretation of the results. It was decided to display the diagrams in the Yourdon standard notation.

2.2 Development Environment: ET++3.0

ET++3.0(Editor Tool kit) is a framework for building interactive applications which edit and display documents which may contain text, diagrams and graphic entities. The resulting application has a look and feel consistent with other ET applications and contains some standard features. Support for collection classes, iterators, text classes and graphic interface objects is provided. Development and debugging aids in the form of class browsers and inspectors are also provided.

The graphic interface will be developed in C++ in the ET environment running under X-windows on SUN workstations connected to the UNIX system of the Department of Computer Science, Concordia University.

2.3 Facilities provided by ET++ for its user applications

The ET framework provides as default a window to the application with a menu bar containing print, load, save, quit, cut/copy/paste and undo/redo commands. These menus and additional menus can be added as required by the application. ET provides the window environment required for the Π-DFD system with a lot of default control behavior defined.

2.4 The Π-DFD graphic interface

2.4.1 Menus provided

File	New / Open / Load / SaveAs / Close / Print / Quit
Edit	none
States	Size / Deadlock State
Equivalence	Check DFD Processes Equivalence
Simulation	Start / Select Next Step / Nb Steps for Random Transition / Select Previous / Quit Simulation

2.4.2 Functionality provided

A graphical representation of the DFD file is to be displayed in the main window when a file is loaded. In the first stage, an additional simulation menu would be provided to allow the user to interactively control a simulation while the diagram would be updated continuously to represent the current state. This will be achieved by interpreting the user mouse input, communicating with CWB and reinterpreting the resulting textual output into the appropriate graphical form. It was decided for this project to represent only flat DFDs i.e. those not decomposed into two or more levels

2.5 File facilities

All the file facilities are to be accessed through the *File* menu selection. There are several *File* options available in the ET system namely, *New*, *Open*, *Load*, *SaveAs*, *Close*, *Print* and *Quit*. The options *Open*, *SaveAs*, *Close* and *Print* have been left with the default behavior of ET. For the purposes of this project the file option *Load* is used to load the diagram and *Quit* to close the application. Initially the system opens with a blank window and the user is free to choose available menu options.

2.5.1 File naming convention

There are two files required to use the diagram display and simulation facility. They are as follows for a translated .TPL file:

0: ((SOURCE_TERMINATOR T1 ((E0,a)) 150 50)

(PROCESS E0 1 ((E1,r)) 150 130)

(PROCESS E1 2 ((E2,s),(E3,v)) 150 210)

(PROCESS E2 3 ((E3,u)) 80 250)

(PROCESS E3 4 ((T2,b)) 150 290)

(SINK_TERMINATOR T2 170 370))

The corresponding translator output gives <filename>.TAB

T1 1 0 150 50 E0 a

E0 0 0 150 130 E1 r

E1 0 0 150 210 E2 s E3 v

E2 0 0 80 250 E3 u

E3 0 0 150 290 T2 b

T2 1 0 170 370

This coded format represents the DFD process names, whether the nodes are terminal or internal, their activity, and the node position on the diagram. It also has pairs consisting of a successive process name and the connecting action name e.g. in line 1 of the above, process T1 is a terminal node, initially inactive, and positioned at point (150,50). It has a successor node E0 which is connected to T1 by action a.

The translator also outputs <filename>.CWB which contains the diagram representation in CCS form which is suitable for the CWB analytical engine. For the previous examples the equivalent .CWB file would contain:

```
bi DFD (T1 | E0 | E1 | E2 | E3 | T2)\{ r, s, v, u}

   bi T1 input . 'a . T1

   bi E0 a . 'r . E0

   bi E1 r . 's . E1 + r . 'v . E1

   bi E2 s . 'u . E2
```

bi E3 v . 'b . E3 + u . 'b . E3

bi T2 b . 'output . T2

For each DFD to be displayed, analyzed and simulated there should be one of each of the .TAB and .CWB files with corresponding filenames. Any errors in the *.TPL* file will be intercepted by the *translator*. It is assumed therefore that the .CWB and .TAB file formats are generated correctly by the *translator.*

2.5.2 File/New

Opens a new application window with a unique title.

2.5.3 File/Open

This option opens a file which has been saved in the ET output format. It is not used for our purposes.

2.5.4 File/Load

This option is used to load and display the diagram to be generated from the two files <filename>.TAB and <filename>.CWB. When *Load* is selected a popup window gives the user a chance to enter a filename which should be a .CWB file. The system will automatically find the corresponding .TAB file, interpret it and display the diagram. At the same time the .CWB file name is passed to the CWB engine.

2.5.4.1 DFD File already loaded

If there is already a DFD file diagram displayed when *File/Load* is chosen then the current diagram will be overwritten by the user's new choice of file and the new file loaded into CWB.

2.5.4.2 CWB Command Interpretation

The CWB command that defines the DFD to be drawn is:

```
clear
if  CWB_File
```

2.5.5 File/Close

This has the ET default behavior. All files associated with DFD that were previously open are closed and the application terminates.

2.5.6 File/Save

This option is *dimmed* for the first version of Π-DFD system.

If the future enhancement of on-screen diagram editing is incorporated, the facility would become available.

2.5.7 File/SaveAs

This option saves the current contents of the window as defined by the default behavior.

2.5.8 File/Print

This option is not used in the first version of Π-DFD system.

2.5.9 File/Quit

This facility allows the user to exit from the Π-DFD System. The process and the CWB process which was opened in association to it is closed.

2.5.9.1 CWB Command Input

CWB command: `clear`

2.6 State checking facilities

The state checking facilities are only available when a diagram is loaded, otherwise they are dimmed.

2.6.1 States/Size

Size information for the DFD is shown in a popup window.

2.6.1.1 CWB Command Interpretation

CWB Command: `size agent`

2.6.1.2 CWB output format

CWB Output is of format

```
agent has N states.
```

and is displayed in a popup window.

2.6.2 *States/Deadlock State*

For the currently loaded DFD possible deadlock states are obtained from CWB and the results shown in a popup window. Deadlock condition is reached when no more observable actions can be performed.

2.6.2.1 Deadlock Input

There is no input since the whole DFD, or the main process of the DFD, is considered for its deadlock state.

2.6.2.2 Deadlock Output

A DFD may have no deadlock states, or it may have one or more deadlock states. These two cases are displayed to the user in a popup window.

2.6.2.2.1 No deadlock

A popup window indicates that the DFD has no deadlock state.

2.6.2.2.2 A number of Deadlock states exist

A scrollable list is displayed listing the sequence of observable transition steps that took place resulting in deadlock state.

2.6.2.3 CWB Command Interpretation

CWB Command: **fdobs Buff_and**

CWB Output will be displayed in popup window:

```
=== a a a a ===>
    ('r.E0 | 's.E1 | 'u.E2 | v.'b.E3)\{r,s,u,v}
=== a a a ===>
    ('r.E0 | 'v.E1 | E2 | u.'b.E3)\{r,s,u,v}
```

- there are two deadlock states,
- there are two *sequences* of observable *transition steps* made to get to the two deadlock states and are represented by:

```
=== a a a a ===>
=== a a a ===>
```

- there are two *StateInformations*, they are given in the data delimited by **()** :

```
('r.E0 | 's.E1 | 'u.E2 | v.'b.E3)
('r.E0 | 'v.E1 | E2 | u.'b.E3)
```

The DFD is updated using the conventions of *StateInformation* in Section 2.8.2.5.

2.7 Equivalence checking

The state checking facilities are only available when a diagram is loaded, otherwise they are dimmed.

2.7.1 Equivalence/Check DFD Processes Equivalence

For any process currently loaded their equivalence can be checked via a menu command and the results shown in a popup window. This will check if two processes are semantically or observationally equivalent. This command prompts the user for another process name. Its equivalence is checked against the main process of the loaded diagram to determine if they are semantically equivalent.

Note that there is currently only one *.CWB file loaded into Π-DFD system at a time and Section 2.10.4 describes a useful enhancement for this option.

2.7.1.1 Input

The mechanism for selecting the process name for equivalence is by popup window where the user can type the process name in a dialogue box.

2.7.1.2 Output

A popup window indicates whether the two processes are equivalent or not If the processes are not equivalent, it also displays the weak modality HML formula distinguishing between the two processes [M 92, p. 19].

2.7.1.3 CWB command Interpretation

The CWB command that checks for observational equivalence is:

CWB command is: `dfobs Process1 Process2 NotEquivalentId`

If the two processes are not equivalent, then the weak modality HML formula distinguishing between the two processes is determined with:

CWB command: `ppi NotEquivalentId`

2.8 *Simulation*

2.8.1 Simulation control

In simulation mode the user is to control the simulation through mouse input and to observe the results of any simulation step through dynamic alterations in the graphical output. Additional simulation facilities for multiple random steps and for displaying the simulation history are provided. The simulation process is taken to be the principal process of the currently loaded DFD.

For mouse driven input of simulation commands, mouse input is to be directly on the action lines immediately preceding and succeeding a potential process to select the next step. If random selection of the next step is desired the user is to input directly onto the potential process. Potential processes and actions as well as active processes will be highlighted as shown in Section 2.9.

Before a simulation session is activated by menu selection, only the *Start* option is shown on the menu. Once the simulation has commenced, the *Start* option is dimmed and the remaining simulation menu options bolded. For this implementation, all options not related to simulation are also dimmed.

2.8.2 CWB Command Interpretation for simulation

2.8.2.1 CWB input

Once a DFD has been selected, the CWB simulation process in CWB needs to be started. The following example explains how this is to be done:

```
E0 = a.'r.E0
E1 = r.'s.E1 + r.'v.E1
E2 = s.'u.E2
E3 = u.v.'b.E3 + v.u.'b.E3
BuffAnd = (E0 | E1 | E2 | E3)\{r,s,u,v}
```

CWB command: **`sim BuffAnd`** is used

`BuffAnd` is defined as the main process in the CWB file and simulation will only be allowable on the main process.

2.8.2.2 CWB Output

The format of the CWB output from **`sim`** command is:

```
Simulated agent:  BuffAnd
```

```
Transitions:

        1: --- a ---> ('r.E0 | E1 | E2 | E3)\{r,s,u,v}
```

There may be several transitions lines output from the `sim` command.

2.8.2.3 Interpreting Potential Action

The potential actions are extracted with the following interpretation of the CWB output:

"`Simulated agent:`" is the current state of the DFD; in this case the DFD is initially and `BuffAnd` is equivalent to `(E0 | E1 | E2 | E3)\{r,s,u,v}`.

"`Transitions:`" indicates that all the remaining lines of the output are possible transitions. They are interpreted as potential actions with definitions as follows:

- each transition line has an *ActionNumber*, the first item on the line before `:` .
- each transition line has an *InputActionLabel* which is contained between -[space] and [space]- for observable actions or between brackets <> for unobservable actions.
- *OutputActionLabel* can be found within the state information in brackets( ) following the >. For process E0, the state is represented as `'r.E0`. Since the state of process E0 would change from the current idle status of `E0`, the action `'r` is the *OutputActionLabel*. Note that if the state of a process has not changed, then the process is simply activated and some more input is pending before the output actions become potential actions.
- *PotentialProcessLabel* is defined as the process associated with *OutputActionLabel*. This is the process that will be affected if the *InputActionLabel* is selected.

- a single *PotentialAction* consists of these elements: (*ActionNumber*, *InputActionLabel*, *PotentialProcessLabel*, *OutputActionLabel*). In this example, the set of *PotentialActions* is `{(1, a, E0, r)}`.
- each data flow diagram object in the potential action has a *potential* status: there is a pair of potential input/output actions and a potential process.

2.8.2.4 Displaying Potential Actions

Each potential process, potential input and output action is displayed on the DFD using the conventions for potential processes and actions described in Section 2.9.3.

2.8.2.5 Interpreting Process Status

Each idle or activate Process is displayed on the DFD using the conventions described in Section 2.9.3.

- each transition line has *StateInformation* for the process which is contained between `(` and `)` ; in this example, the StateInformation is;

  `('r.E0 | E1 | E2 | E3)`
- the *StateInformation* contains the *ProcessState* for each process in the DFD and which are separated by | ; in this example, the process states are `'r.E0 , E1 , E2 , E3`
- from the *ProcessState* it is possible to determine the status of the process
  - any process in a *ProcessState* that has only the process name has an *idle* status in this example, the idle processes are E1, E2 and E3,

- any *ProcessState* that has more than the process name has an *active* status: in this example, the active process is E0.

2.8.2.6 Single Step through a DFD

A Single Step through a DFD means a single transition on an action: it is sent from one process and received by the connected receiving process. The next potential steps are all the possibilities for a single transition on the DFD, given the current state of the DFD. The DFD will use conventions described in Section 2.9. Notably, the next potential steps are identified on the DFD by highlighting of the potential input and output actions from a similarly highlighted potential process The next potential step can be input using the mouse by first clicking on the potential input action then clicking on the potential output action. Where the next step is to be randomly chosen by the system the user clicks directly on the potential process. The user can also choose the next step via the menu if desired.

2.8.2.6.1 Interpretation for a single step

Once the potential action has been selected, the DFD goes through the transition by activating the potential action. Here is the description of CWB command that does this.

CWB command: `ActionNumber`

Each potential action has an *ActionNumber* as described in Section 2.8.2.3.

2.8.2.6.2 Interpretation for a Random Single Step

For a random choice of a single step, the user will mouse click on the transmitting process and an action number will be randomly selected from those available.

CWB command: **ActionNumber**

Note: CWB random command cannot be used in this case because it randomly selects any potential action within the data flow diagram and does not necessarily select a potential action that affects the process that the user has selected.

2.8.3 Interpretation of CWB Output

After selecting the action, the DFD is updated according to the new status of the processes. This new state will result in a new set of potential actions; only the potential actions will be displayed as such. For a user selected next step and a randomly selected next step, these have the same CWB Command Output.

2.8.3.1 CWB command output interpretation

CWB command output is:

```
--- a --->
Simulated agent:
   ('r.E0 | E1 | E2 | E3)\{r,s,u,v}
Transitions:
1: --- t<r> --->   (E0 | 's.E1 | E2 | E3)\{r,s,u,v}
2: --- t<r> --->   (E0 | 'v.E1 | E2 | E3)\{r,s,u,v}
```

Where:

- the *TransitionStep* made to get to state **Simulated agent** is represented by **---a --->**
- the *StateInformation* is given in the data after **Simulated agent:** and delimited by **()** is **('r.E0 | E1 | E2 | E3)**
- the set of next *PotentialActions* is defined in Section 2.8.2.3 and is **{(1, r, E1, s), (2, r, E1, v)}**

The DFD is updated using the conventions in Section 2.8.2.4 and Section 2.8.2.5, using the *StateInformation* and the set of *PotentialActions*.

CWB uniquely identifies each potential action by the ActionNumber. In the simulation, each potential action is uniquely identified by the pair of input and output actions.

2.8.4 Simulation/Nb Steps for Random Transition

On user choice of the menu option Simulation/Nb Steps for Random Transition followed by input of the number of steps required, the Π-DFD system will automatically make a number (Nb) of transition steps through the DFD. Each transition step, however, will be randomly selected. The resulting state of the DFD after these Nb transitions, and the potential actions from the resulting state is displayed in a popup window showing the CWB text output. The random mode of activation from the menu is the only means of making more than one transition step.

2.8.4.1 CWB command interpretation

Once the potential action has been selected, the DFD goes through the Nb transitions by use of the command:

CWB command: **`random Nb`**

where **`Nb`** is the number of transitions.

2.8.4.2 CWB Command Output

As an example, using **`random 2`**, the CWB command output is:

```
--- a --->
--- t<r> --->
Simulation complete.

Simulated agent:
   (E0 | 's.E1 | E2 | E3)\{r,s,u,v}
Transitions:
1: --- t<s> --->   (E0 | E1 | 'u.E2 | E3)\{r,s,u,v}
2: --- a --->      ('r.E0 | 's.E1 |E2 | E3)\{r,s,u,v}
```

Where:

- the 2 *TransitionSteps* made to get to state **`Simulated agent`** are represented by

```
--- a --->
--- t<r> --->
```

- the current *StateInformation* is given in the data after **`Simulated agent:`** and delimited by **`()`** is **`(E0 | 's.E1 | E2 | E3)`**

- the set of next potential actions is defined by Section 2.8.2.3 and is:

  `{(1, s, E2, u), (2, a, E0, r)}`

2.8.5 *Simulation/Select Previous*

This menu input allows the user to determine what transition steps have previously taken place in the current simulation session and for these steps to be displayed in text form in a scrollable popup window.

The user can select one of these DFD status and the DFD will return to, and be displayed in, the selected status.

2.8.5.1 CWB command input

There are two commands used for this option. The first command displays the possible DFD status. The second command returns to the selected DFD status.

CWB command: **`history`**

CWB command: **`return ItemNb`**

2.8.5.2 CWB Command Output

The output from CWB is a list of transition steps that have taken place in the simulation session.

Continuing with the example, the CWB command output is:

```
0:  Buff_and --- a --->
1:  ('r.E0 |E1 |E2 |E3 )\{r,s,u,v} --- t<r> --->
```

```
2:   (E0 |'s.E1 |E2 |E3 )\{r,s,u,v}
```

This information is displayed in a scrollable window.

`return ItemNb` is performed for selected ItemNbs 0, 1 or 2

2.8.6 Simulation/Quit Simulation

Quits the simulation session and the data flow diagram returns to an idle state with all processes and action lines displayed as inactive.

CWB Command: `quit`

2.9 Graphic Display Conventions

These are the conventions used to display the status of the different types of processes and actions in the DFD diagram. The graphical display uses the Yourdon Standard Method of displaying the objects in the DFD.

2.9.1 Process display status

There are three different conventions for graphic display of the status of processes which may be internal processes or source/sink terminators:

- potential: a process with a potential status has a pair of potential input/output actions associated with it (see Section 2.8.2.3).
- active: a process with an active status has at least one active input or output action associated with it, but no potential actions.
- idle: a process with an idle status has no active, or potential, input or output action associated with it as described in Section 2.8.2.5.

2.9.2 Action display status

Conventions for graphic display of actions:

- potential: a potential action is an action that is a member of a pair of potential input/output actions within the data flow diagram (see Section 2.8.2.3).
- idle: an idle action is an action that is not potential.

2.9.3 *Graphic display conventions*

2.9.3.1 Processes

Conventions for processes:

	idle	**active**	**potential**
terminal			
internal			

2.9.3.2 Actions

Conventions for action lines:

idle	**potential**
⟶	⟶

2.10 Future Enhancements

There are many features that could be added and should be considered as possible extensions to the product in the future.

2.10.1 Multiple levels of decomposition

Diagrams with multiple levels of hierarchical decomposition could be handled.

2.10.1.1 Input

2.10.1.1.1 Menu Input

An input mechanism would be provided to identify the level of refinement for either an individual process or for the entire data flow diagram.

Providing the option of defining the level of refinement for an individual process makes it possible to display the process at a different level of refinement from the rest of the data flow diagram. For example, if Process1 has up to three levels of refinement while Process2 has none, the user could display the lowest level of refinement for all processes.

2.10.1.1.2 Mouse Input

Mouse click on the process that requires display of refinement one level lower than current level of refinement.

2.10.1.2 Output

A new window is opened displaying the DFD of the specified process at the required refinement level.

All the facilities (menu options) need to be available since normally level 0 is not very interesting and we need to do queries on lower level processes.

2.10.1.3 Error

If the user clicks on a process that does not have a refinement, an error window could appear indicating that there is no more refinements for this process.

2.10.1.4 Closing a refinement

A refinement is closed by closing the DFD window.

2.10.1.5 Further considerations

Here are some other considerations that have to be taken into account for this option.

- how feasible is it to do a simulation on parent DFD;
- should changing the refinement level be provided while in simulation mode, and if so, determine how to inform the CWB of the additional information; and
- the coordinates for a process within a window is currently defined in the tuple representation of the data flow diagram; in order to display a process in different refinement levels, the mechanism for determining the coordinates has to be modified

2.10.1.6 Display of hierarchical levels using multiple windows

There could be display of different levels of DFD using multiple windows with CWB interpreting each process at the lowest level. In order to display a window with a higher level DFD (level 0, for example), it is necessary to give CWB only the processes in level 0. There is an example in [BGST 95A, Figure 3]: this shows two different decompositions of a system where in the second, D is at level 1 with no refinement. It would be possible to display only level 1 in a window and allow simulation only of Q11 and Q12 at level 1 by not giving CWB information about the decomposition of Q11 and Q12. Similarly, if we wanted to display level 2, we could display level 2 in a separate windows by giving CWB information level 2 data defining Q11. However, this mechanism does not link all the processes together. If wanted to display level 2, the current mechanism for defining process using CCS would make CWB "think" that process1_1 is at level 2.

2.10.2 Simulate a subset of DFD

A facility could be added to simulate only part of a DFD by allowing textual input of the subprocess to be simulated.

2.10.3 Derivation Commands

A separate menu for derivation commands such as to display observable actions/processes reachable, transitions from that process etc.

2.10.4 Comparison between two or more data flow diagrams

Equality comparisons and differentiation between the main processes of different DFDs This facility requires that the system have separate labels for the display and for CWB commands. This is because CWB redefines a process, if it is defined, every time there is a statement with the same process label.

2.10.5 On-screen diagram editing

It would be possible to allow the user to reposition the processes of the DFD.

There are many implications to the Π-DFD system. With this extension, it is necessary to add the following functionality:

- add the editing facilities: add all types of items needed in DFD, move, copy,
- read the tuple representation data from the diagram; this information can be used by the translator to generate CWB code and CCS equations;
- (File/Save) facility would have to save the new process data and its position, as well as the state of the processes; and
- (File/Close) would have to verify if the position of any objects in the DFD has changed. If there are any changes to the data flow diagram, need to add DFD has changed dialogue and allow the user to save the changes as in (File/Save) before closing the file.

2.10.7 Different DFD standards

It would be possible to give the user the flexibility of using different data flow diagram standard notations such as:

- DeMarco's
- Yourdon Structured Method
- Sommerville's
- Rumbaugh
- Ghezzi

2.11 The look of the interface

The typical window interface would has a decorated window and pull-down menus.

Figure 2.11.1 shows the States pull down menu activated with a display of the DFD in an *idle* status (all processes are idle).

Figure 2.11.2 shows the DFD after potential actions have been selected. The processes and actions are displayed with the conventions in Section 2.9.3. An example of an active process is *E0*. There are two potential actions associated with the potential process *E1*; they both have potential input action *r* with either potential output *s* or *v*. The potential numbers are not needed nor displayed on the data flow diagram.

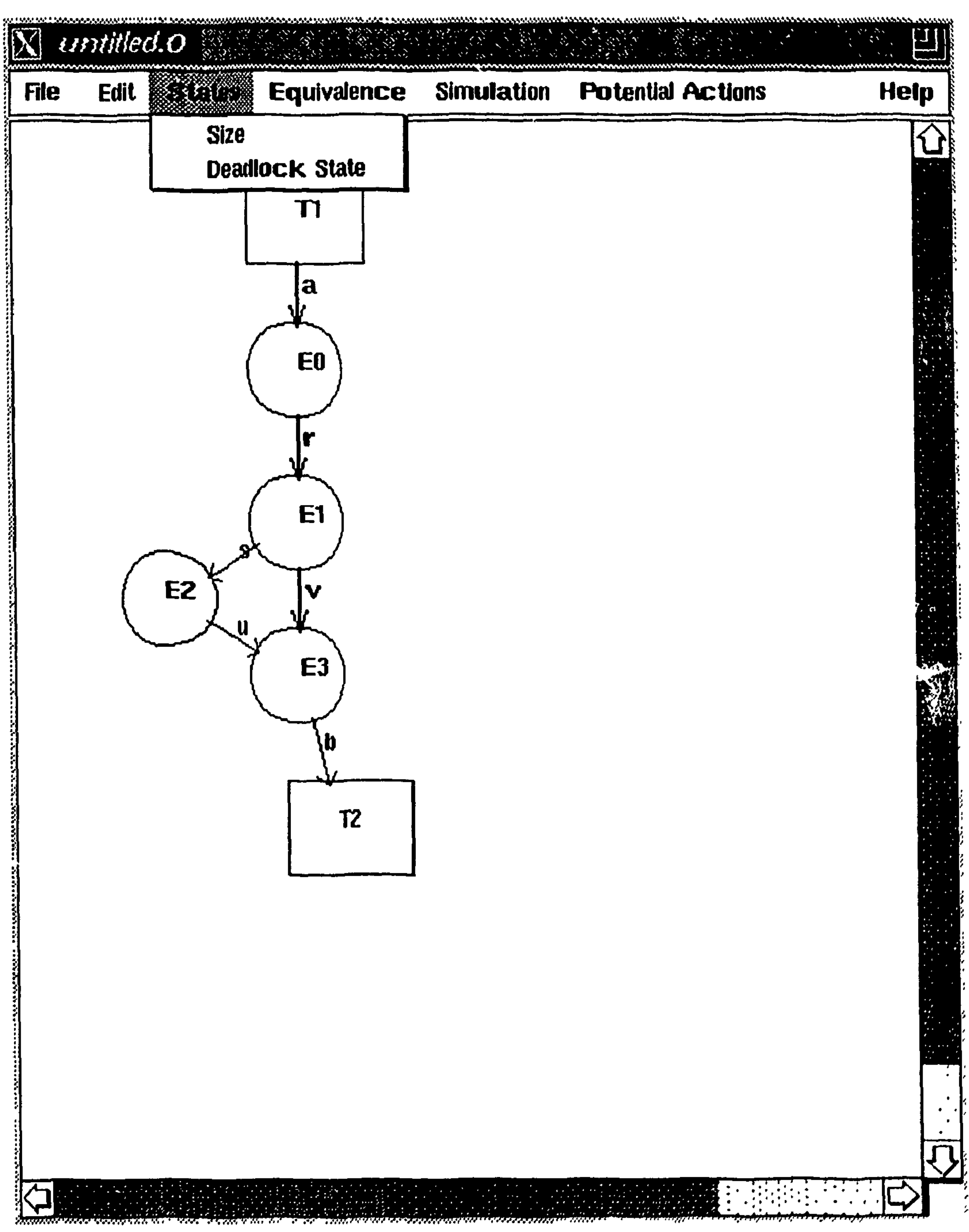

Figure 2.11.1: Π-DFD Graphic Interface -- idle data flow diagram

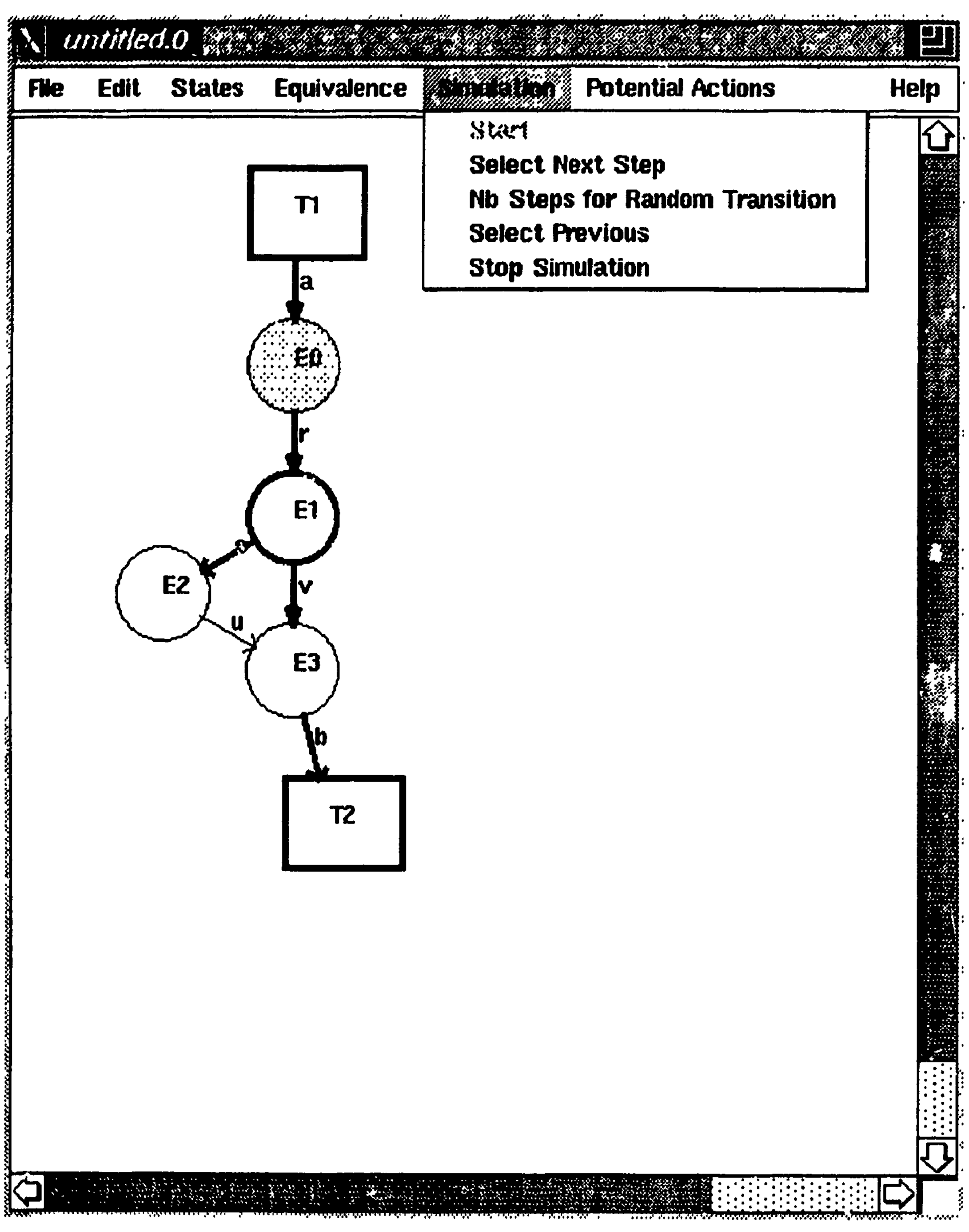

Figure 2.11.2: Π-DFD Graphic Interface -- simulation data flow diagram

3 Design

3.1 Principal Design Diagrams

This section contains the design diagrams associated with the project. This includes the event diagrams, architecture diagrams, and class generalization and aggregation class diagrams.

The relationships between the main components of Π-DFD Graphic Interface are described in Sections 3.1.1 and 3.1.2.

The DFD Application has been divided into two major subsystems: the User Interface and the CWB Interface. The relationship between the classes in each of these subsystems are described in diagrams in Sections 3.1.3 and 3.1.4. All the classes are described in details in Sections 3.2 and 3.3.

3.1.1 Global Event Diagrams

Figure 3.1.1.1 shows the interactions between the user and the principal agents of the system; specifically between the user, the ET window, the translator, the DFD application, the CWB process and the file system. A top level of interaction is shown.

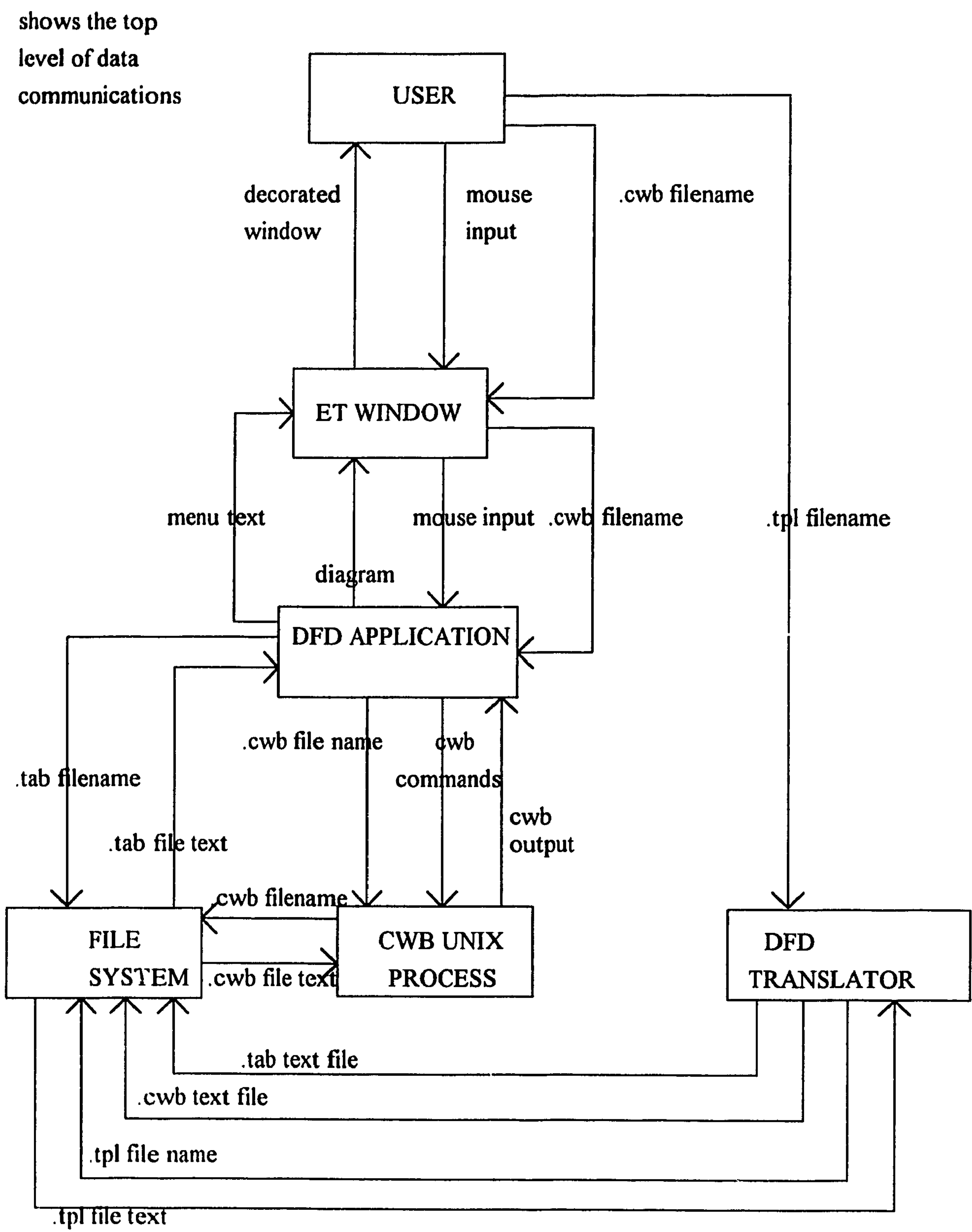

Figure 3.1.1.1: Global Event Diagram (1)

Figure 3.1.1.2 shows more detailed events between the user, the ET window and the two parts of the DFD Application (the DFD User interface and the DFD CWB interface).

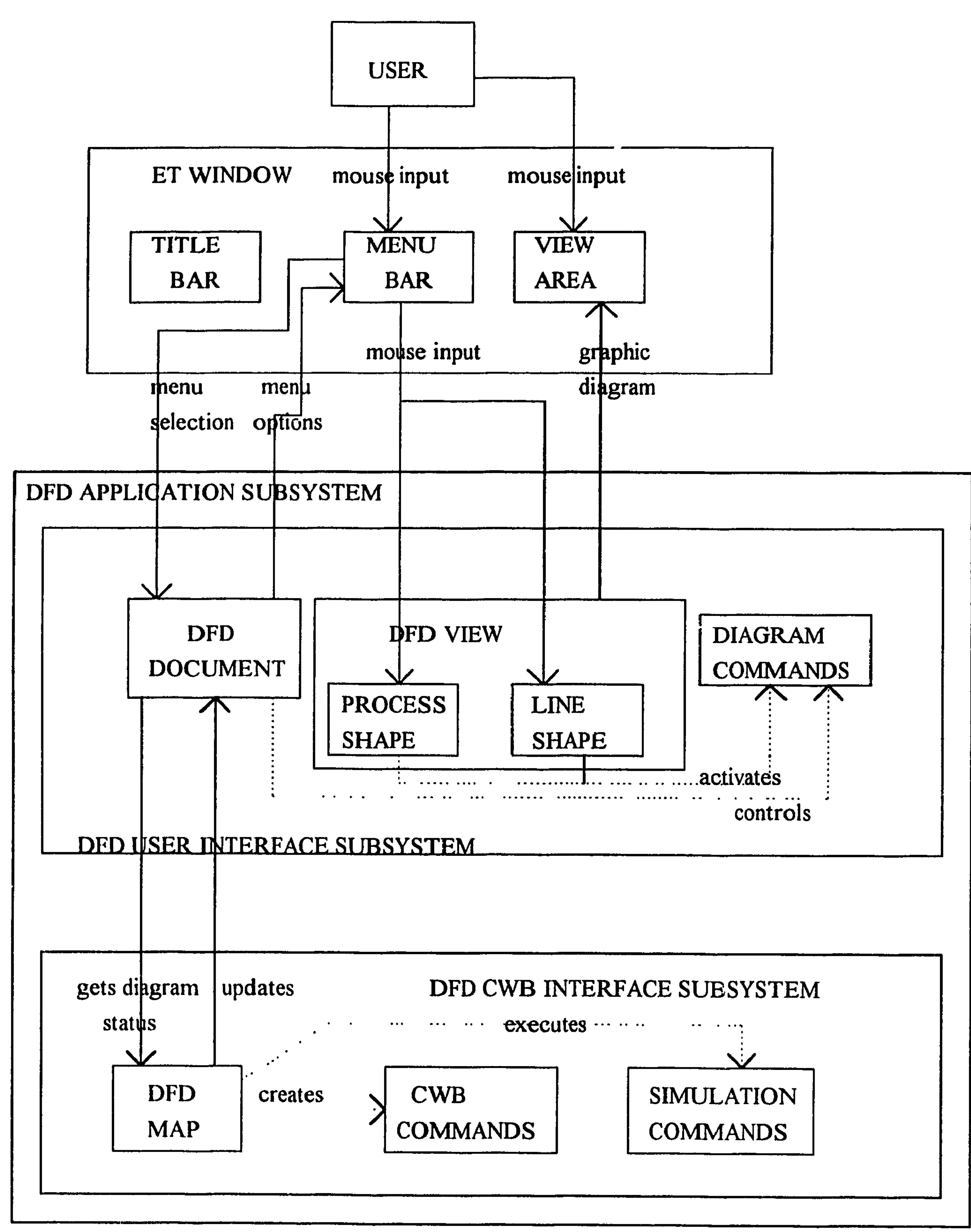

Figure 3.1.1.2: Global Event Diagram(2)

3.1.2 Architectural diagrams

Figure 3.1.2.1 shows the system topology i.e. the interdependence of the main subsystems and their hierarchical relationship.

Figure 3.1.2.2 shows a further breakdown of the major subsystems and the flow of data between them.

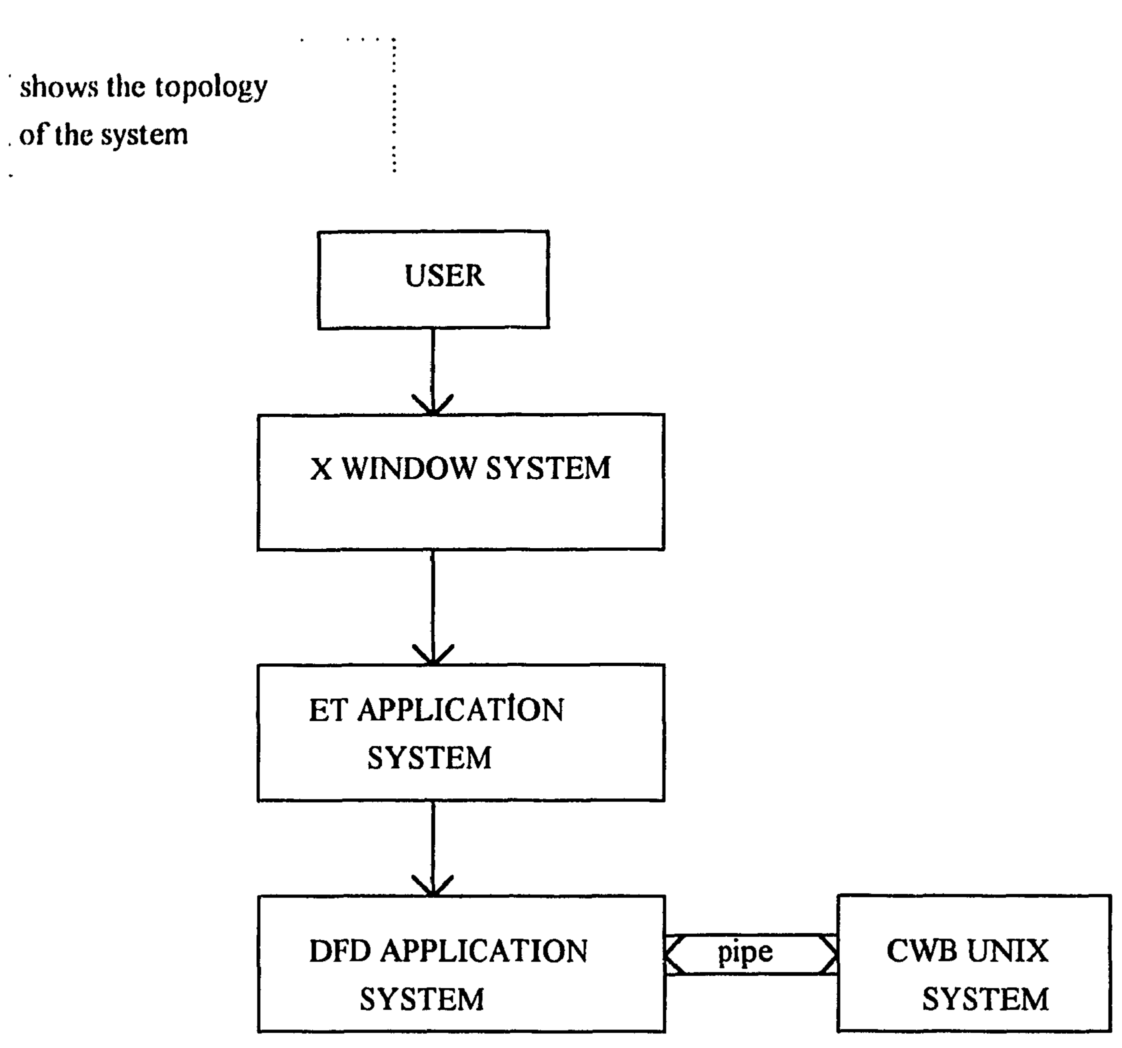

Figure 3.1.2.1: Top Level Architecture

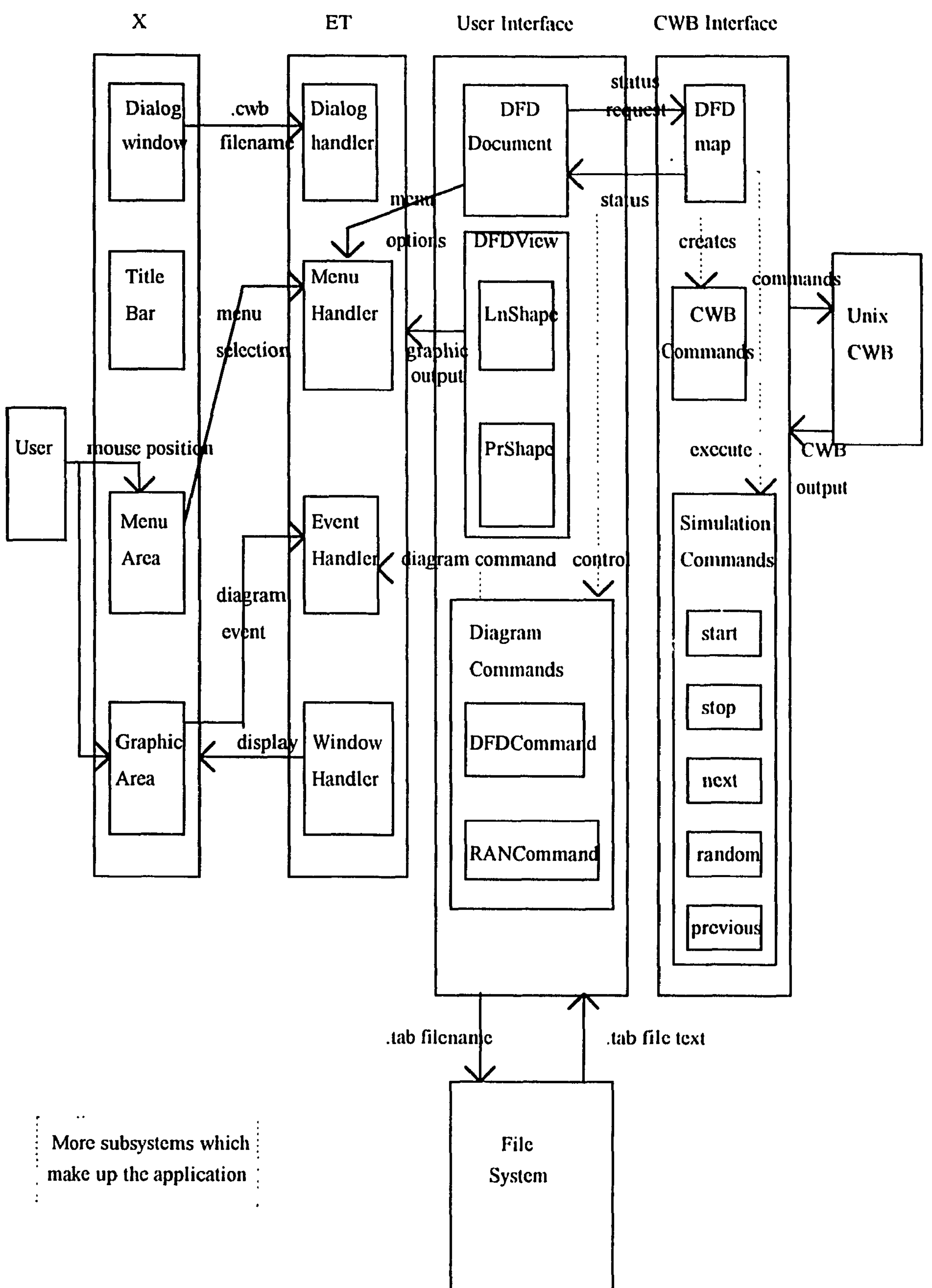

Figure 3.1.2.2: Architecture -- Subsystems

3.1.3 Class diagrams for DFD Application User Interface

Figure 3.1.3.1 gives an overview of the relationship between the document, the view and the application.

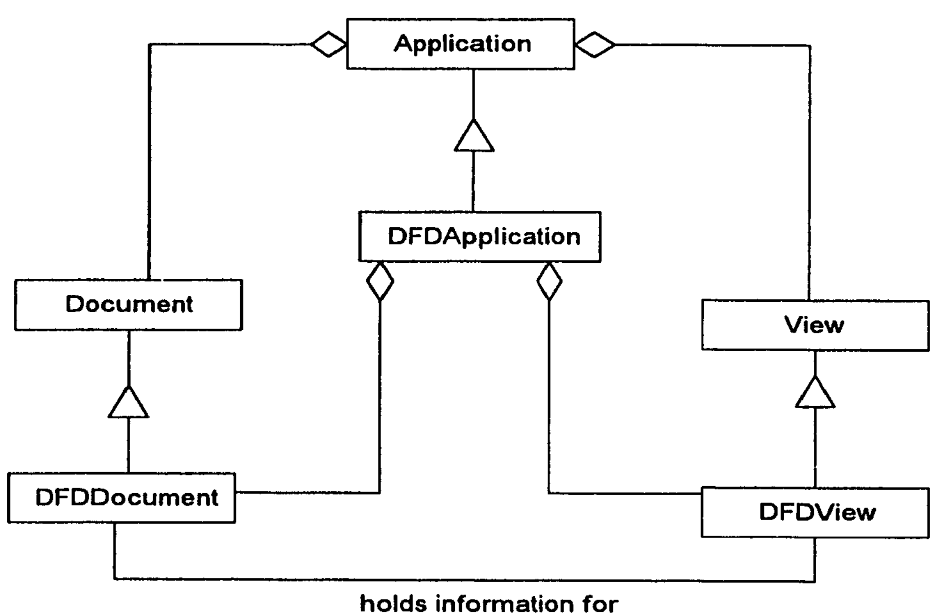

Figure 3.1.3.1: Overview class diagram

Figure 3.1.3.2 is a class diagram that shows how objects representing the processes and actions (dprocess and aline) have VObjects (class which can appear in the view) associated to them. The Lnshape and Prshape objects are based on a generic Shape

object so are collected into one sequence collection. This sequence collection is associated with the View and the information contained in it is used to redraw the DFDView window.

shapes

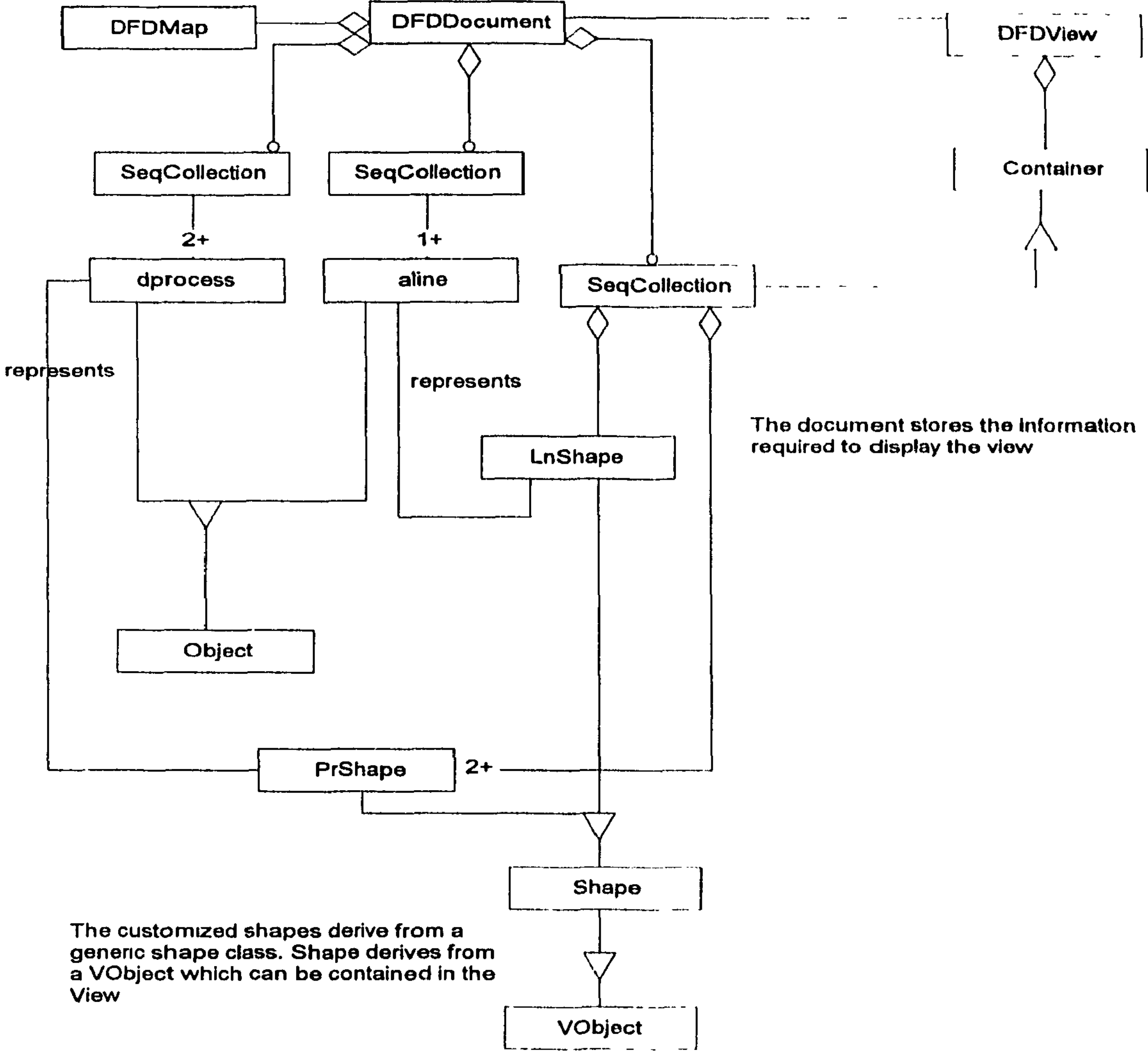

Figure 3.1.3.2: Classes for generation and display of diagram shapes

Figure 3.1.3.3 shows the diagram command classes RANCommand and DFDCommand. These are based on the generic ET Command class. If DoLeftButtonDownCommand is not defined in the View class and is redefined in the view objects LnShape and PrShape, then the general command mechanism is inherited for those shape classes. In this way, in order to make the system respond to clicking on a shape, there has to be an associated user defined command (RAN or DFD) which will be activated when there is a left mouse click on it.

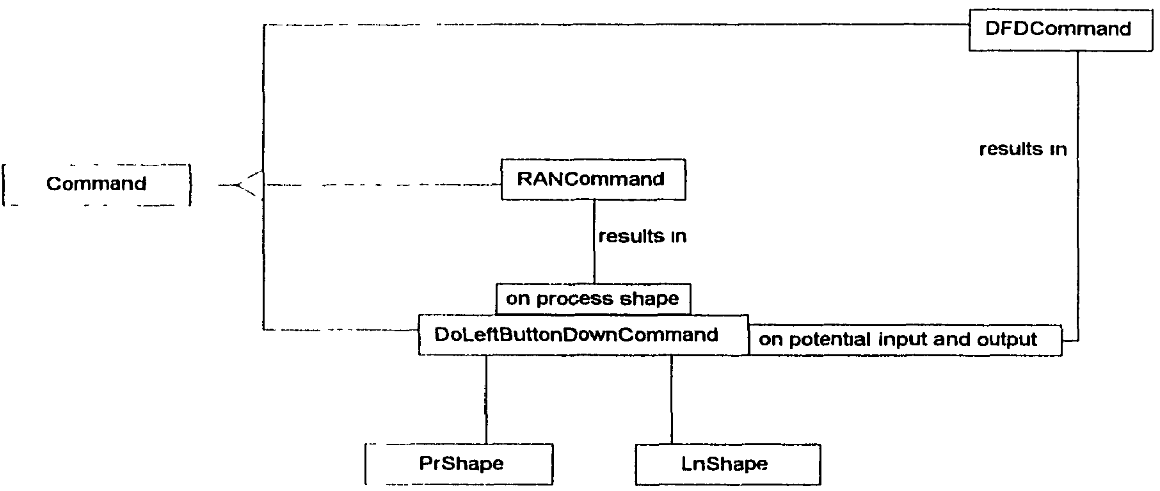

Figure 3.1.3.3: Classes for the diagram commands

3.1.4 Class Diagrams for CWB Interface

This section describes the class diagrams for all the classes grouped in the Cwb Interface. The Cwb Interface is divided into 5 different groups.

1) *Communication with CWB process*: CwbComm is a class that represents the CWB process within the application; it does low level input/output to the CWB process.

2) *Status of the DFD*: ProcessStatus and DfdStatus are classes that represent the statuses of the elements in the data flow diagram. PotentialAction represents the actions that can be made on the DFD while in simulation mode. This information is displayed on the DFD and can be displayed in textual format to the user.

3) *Interface*: DFDMap is the class that provides the interface between the Cwb Objects and the rest of the application.

4) *Analysis Commands*: CwbCommands are commands that are used to *analyze* the DFD without modifying the status of the DFD.

5) *Modifying Commands*: SimCommands are commands that are used to *modify* the status of the DFD.

3.1.4.1 Communication with Cwb Process

CwbComm provides input and output to the CWB process. It is held by DFDMap (see Figure 3.1.4.3.1). It's functionality is described in detail in Section 3.3 1

3.1.4.2 Status of the Dfd

The status of the data flow diagram is represented with these objects. The *current* or most recent status includes the status of the processes and actions in the data flow diagram and are called ProcessStatus. The status of all the elements in the data flow diagram is stored in DFDStatus. When a data flow diagram is in simulation mode, it has potential actions, represented with PotentialActions. These three statuses are held by the DFDMap and their class relationship is shown in Figure 3.1.4.3.1.

To conform with the ET++ format, there are classes that inherit the TextItem class that is used to display text. This can be seen in the Figure 3.1.4.2.1.

3.1.4.3 Interface between Application and CWB

DFDMap class separates CwbInterface from the rest of the application (see Figure 3.1.4.3.1). It starts the CwbProcess, creates all CwbCommands and SimCommands, updates and holds all status information for the DFD, and tells the DFD Diagram that a change in status has occurred (see Figure 3.1.4.5.3).

The DFDMap holds all the status information for the DFD which is generated each time there is a change made through a SimCommand. It contains one current DFDStatus (list of status for each process in the DFD). It also contains 0 or more Potential Actions. A Potential Action has an inputAction, and outputAction and a processName. Each Potential Action has the resulting DFDStatus. This is the next status if the potential action is selected.

DataFlow Diagram Status Classes Generalization

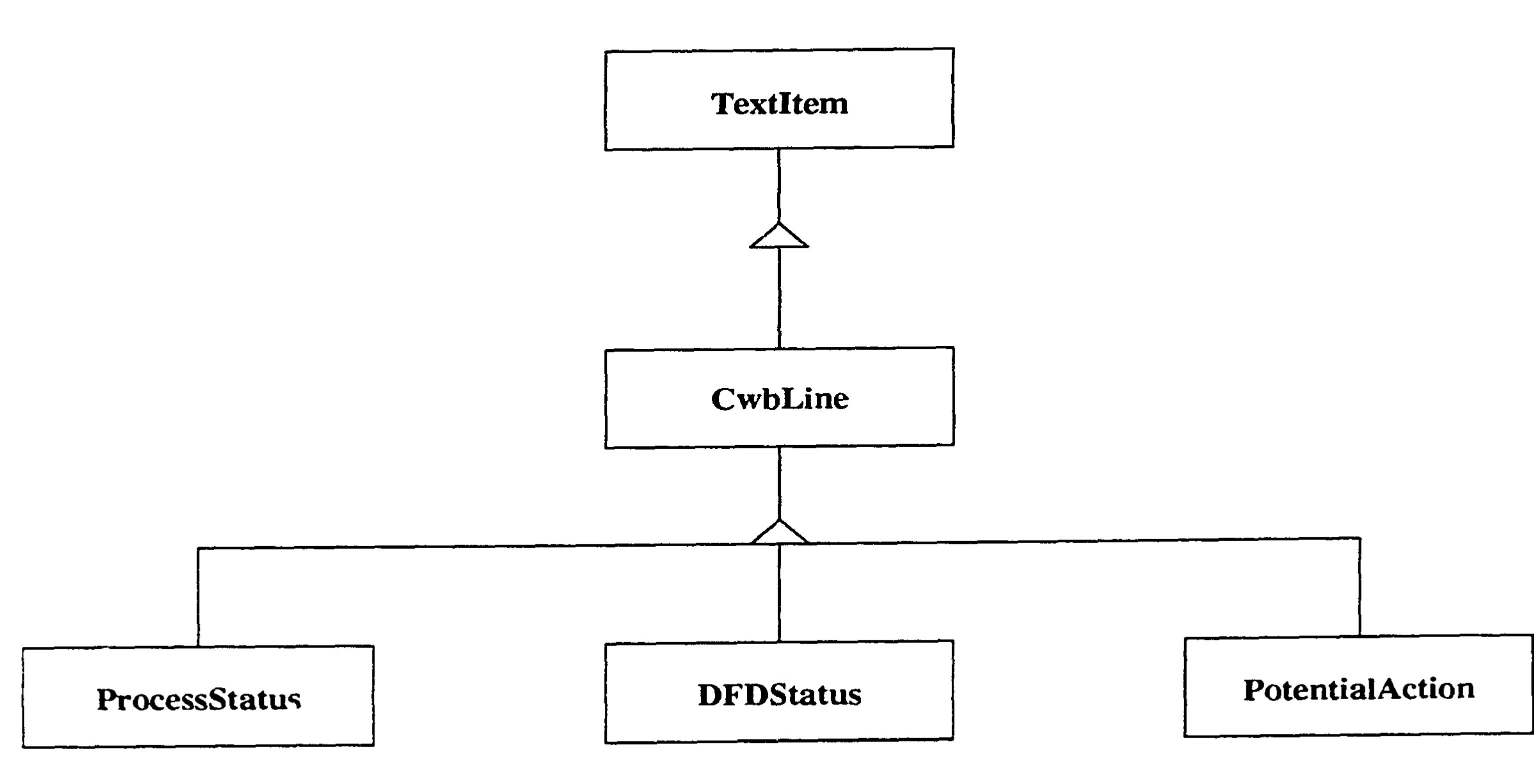

Figure 3.1.4.2.1: DataFlow Diagram Status Classes Generalization

DFD Map Class Aggregation

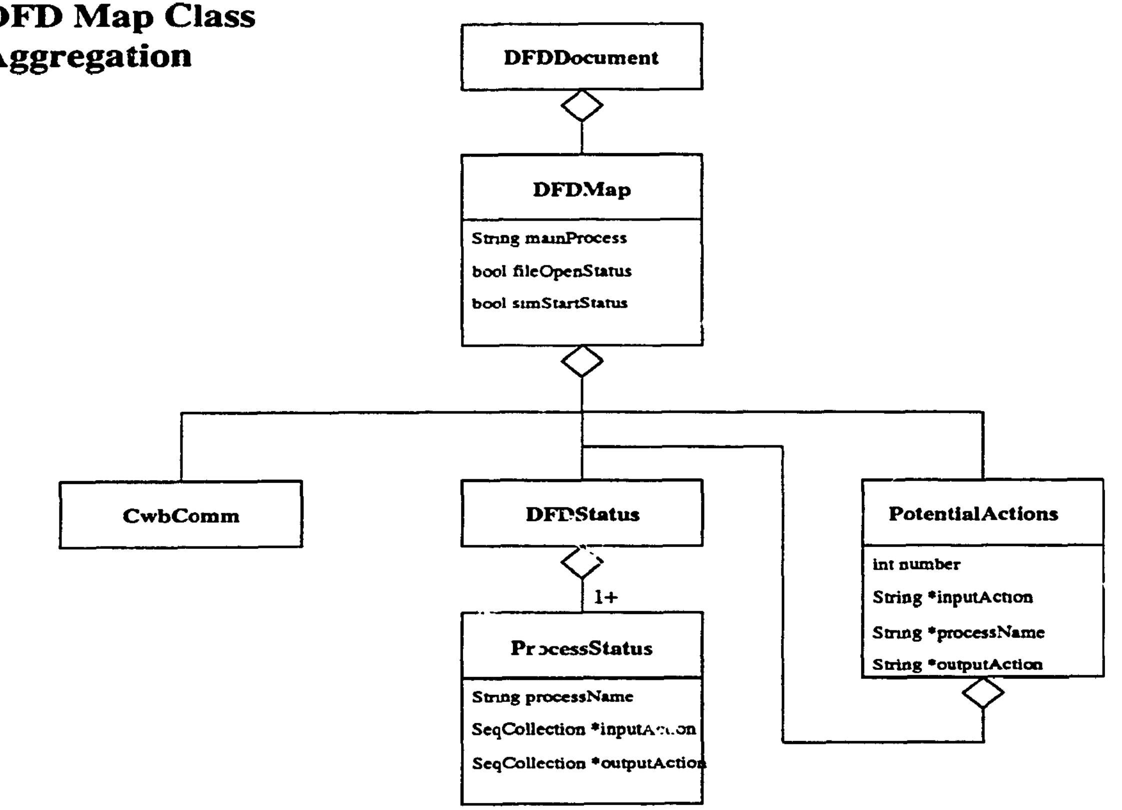

Figure 3.1.4.3.1: DFD Map Class Aggregation

3.1.4.4 Commands that interaction with Cwb Process

The CwbCommands are used by the following menu items: File/Load, States/Size, States/Deadlock State, Equivalence/Check DFD Processes Equivalence. The SimCommands are used by the following menu items: Simulation/Start, /Select Next Step, /Nb Steps for Random Transition, /Select Previous, /Quit Simulation.

All of these commands have similar type of functionality: they are responsible for sending the command to the CWB process and for receiving and interpreting the output from the CWB process. The CwbCommand Class is the base class for the SimCommand. Figure 3.1.4.4.1 shows all the commands that are implemented. For each menu item described above, there is a corresponding class that is instantiated to execute the command. Only the leaves of the class tree are instantiated.

CwbCommands is a group of classes that interact with the CWB process outside of the simulation mode. These are the leaf subclasses of CwbCommand class: FileOpen, Size, Equivalence and Deadlock. Each CwbCommand sends its command to the Cwb Process and does minimal interpretation of the Cwb process output (some parsing and translation) and displays a response to the user.

SimCommands is a group of classes that interact with the CWB process when in simulation mode These are the leaf subclasses of SimCommand class: Start, Stop, Next, Previous, Random. Each SimCommand sends its command to the Cwb Process The interpretation for SimCommands is more complex and the Cwb process output is used to generate the DFDStatus and PotentialActions of the DFD.

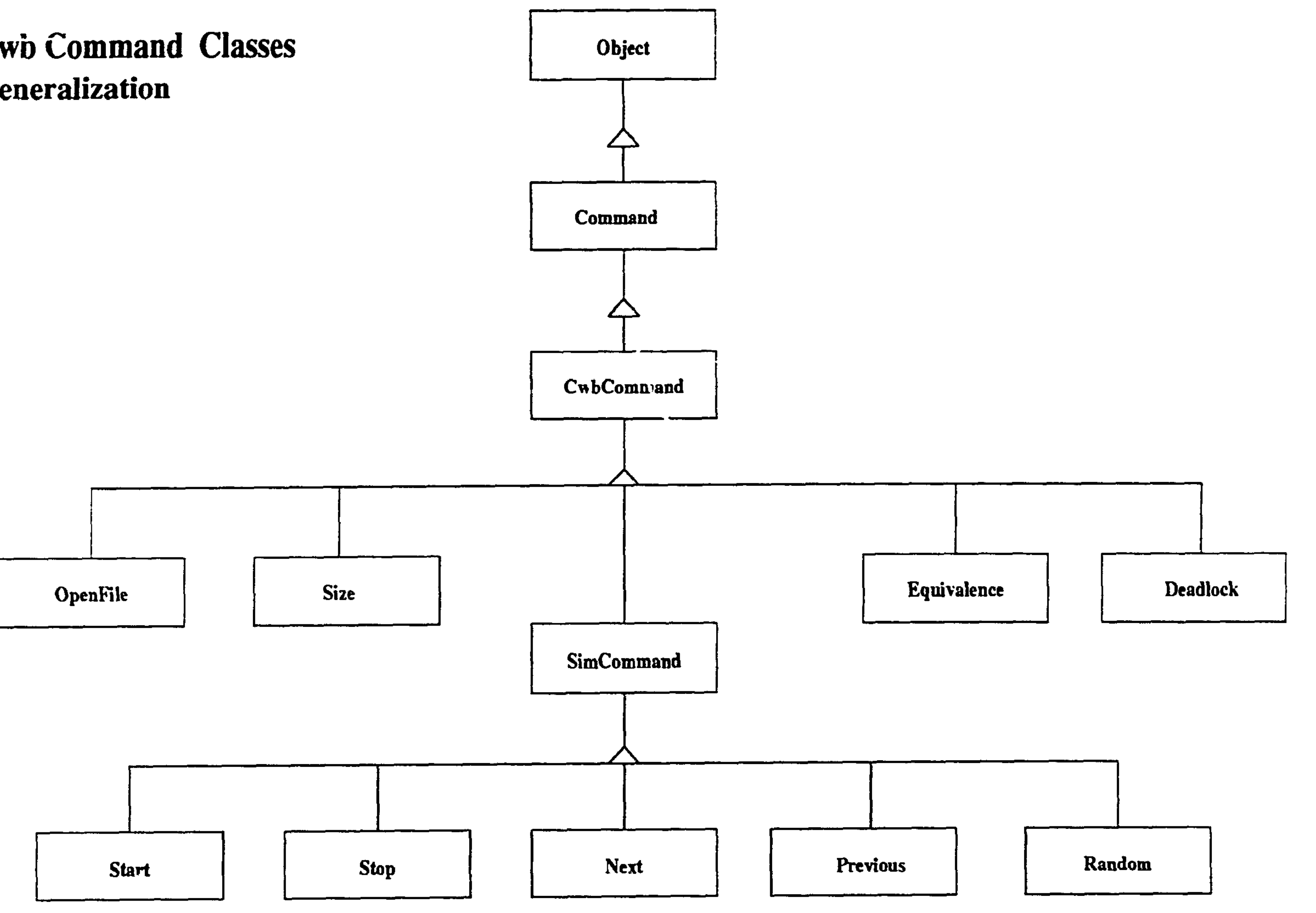

Figure 3.1.4.4.1: Cwb Command Classes Generalization

3.1.4.5 Command Events

In ET++, the commands are first created when the user makes a request to the application through either a menu item, or through the mouse by selecting on the display of the DFD diagram. There are two types of low level commands: CwbCommands and SimCommands. Figure 3.1.4.5.1 shows how these two types of commands are created.

Figure 3.1.4.5.2 describes the time (left to right) relationship and the relationship between the objects used for a CwbCommand. The CwbCommand is first created through the DFDMap (seen in Figure 3.1.4.5.1). The CommandProcessor then sends the message DoIt to the command. This CwbCommand is actually one of the following commands: Size, FileOpen, Deadlock or Equivalence. The CwbCommand uses the instantiation of CwbComm to Send and Receive the command. The command places the result in a CwbLine and displays this to the user in a dialog.

Figure 3.1.4.5.3 describes a similar event pattern to CwbCommand Events except that SimCommand has to update the status information. It does this by telling the DFDMap to Update the status.

DFD Events: User Input

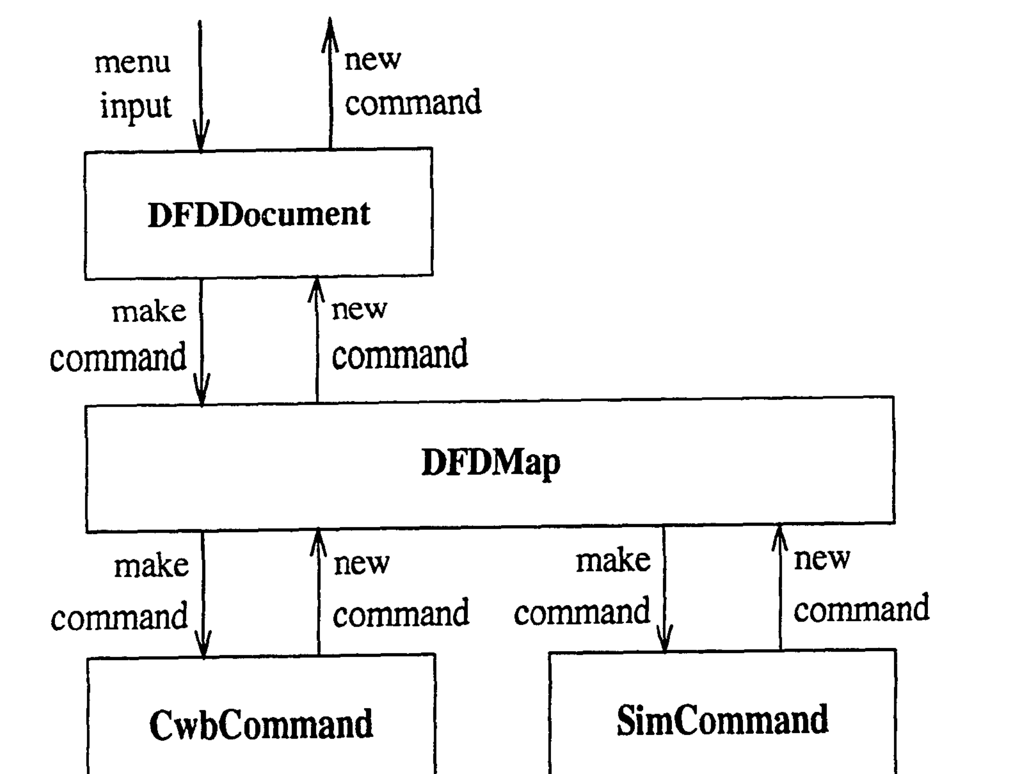

Figure 3.1.4.5.1: DFD Events: User Input

CwbCommand Events

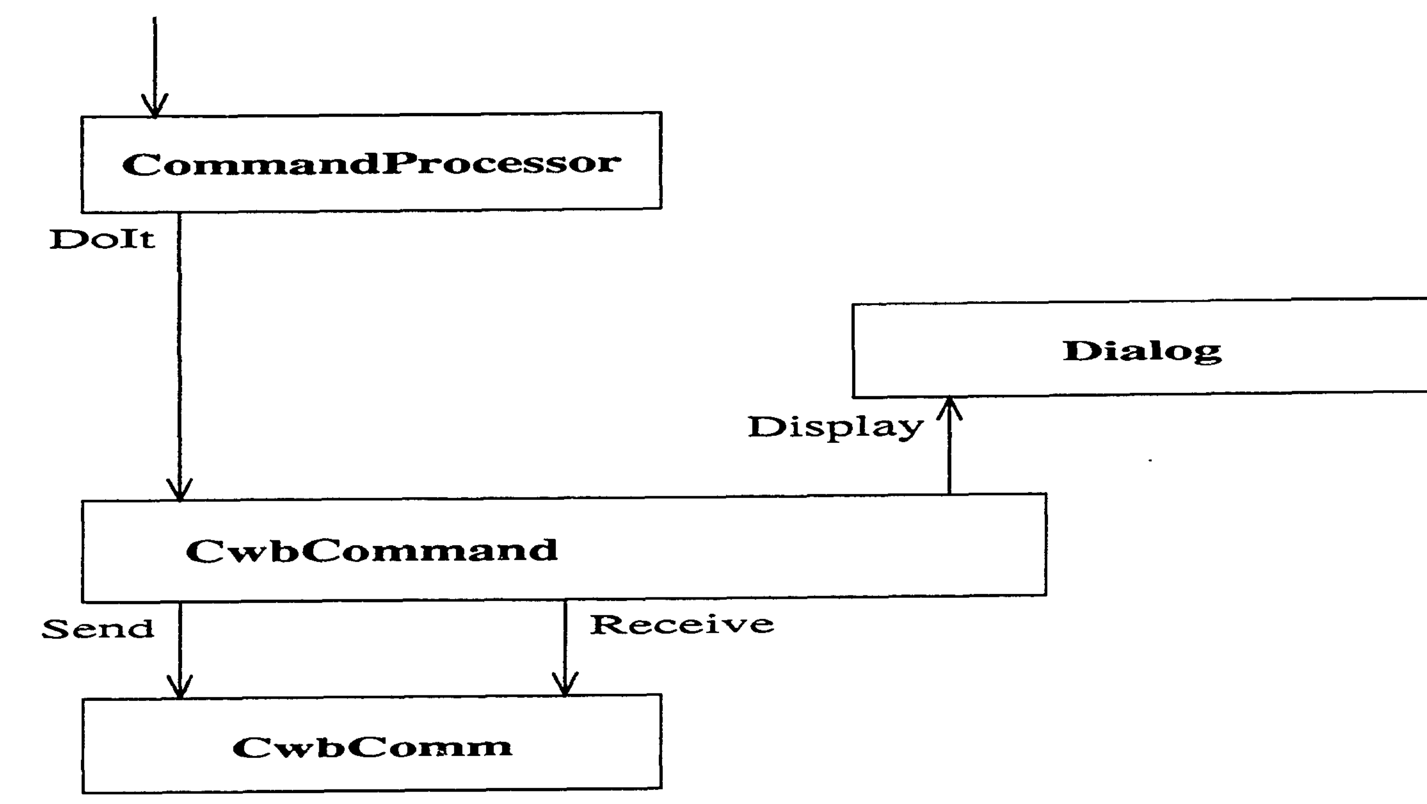

Figure 3.1.4.5.2: CwbCommand Events

SimCommand Events

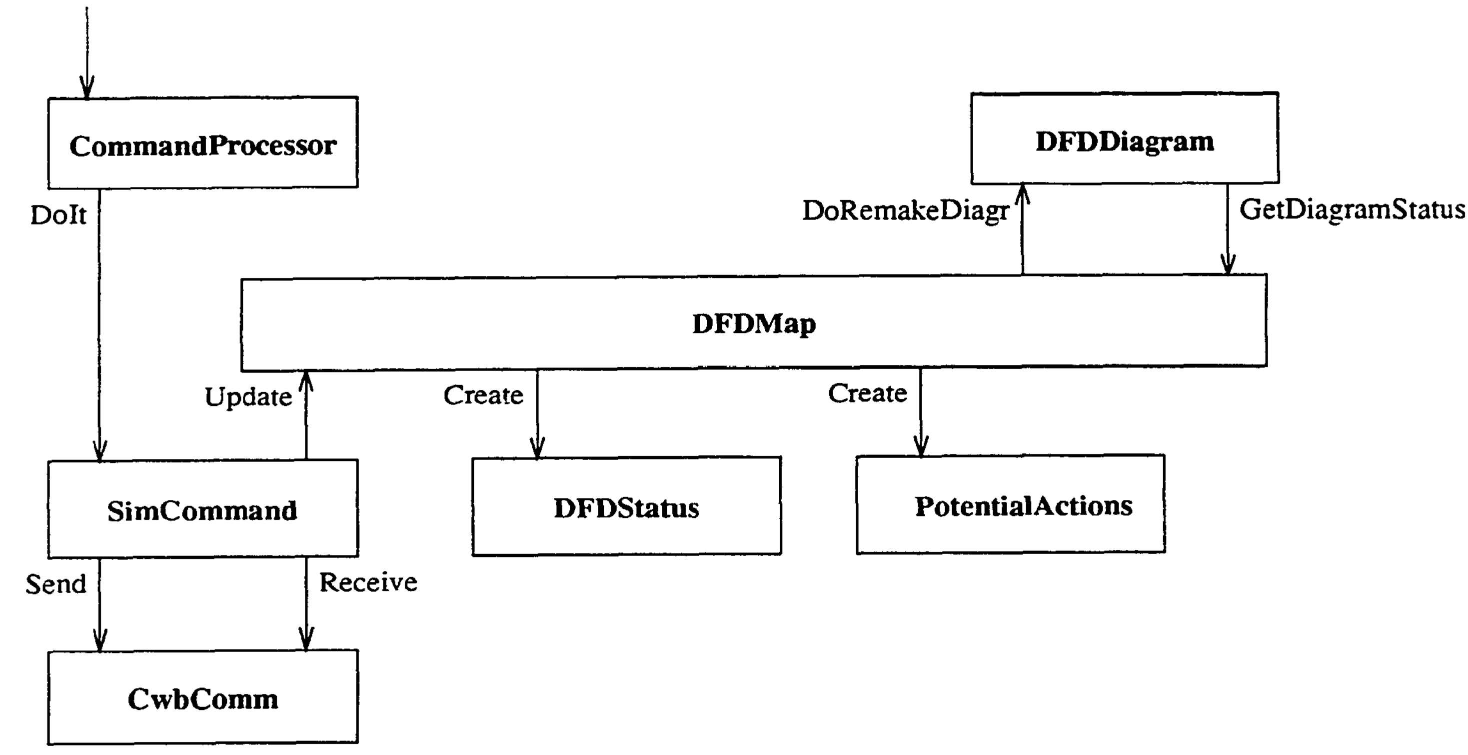

Figure 3.1.4.5.3: SimCommand Events

3.2 Design: User interface

The Document, the View, the Diagram Objects and Simulation Commands

This section describes the classes which relate to the display of the dataflow diagram which is currently under consideration by the user. The classes enable the user to control the analysis and simulation of the diagram using the CWB tool.

3.2.1 DFDDocument Class

The **DFDDocument** class holds and manipulates the data concerning the dataflow diagram which will be displayed in the view. It also defines how the view window is to be redrawn after a change in the status of the window contents. A further function of the document is to define the menu options to be displayed, their status (whether bolded or grayed available/not available for selection) and the appropriate actions to be taken when an option is selected.

Superclass

Document

Attributes

char labl	holds the label of the input action selected by the user in a simulation

char lab2	holds the label of the output action selected by user in a simulation

Private Methods

void *EnableSimItems*(Menu *m, bool b)

Enable/disable menu items *m* which are required in simulation mode depending on boolean value *b*.

void *EnableNoSimItems*(Menu *m, bool b)

Enable/disable menu items required in non simulation mode depending on boolean value *b*.

void *EnableNoSimEqItem*(Menu *m, bool b)

Enable/disable equivalence menu item depending on boolean value.

Public methods

DFDDocument(void): Document(cDocTypeDFD)

DFDDocument constructor.

~DFDDocument(void)

DFDDocument destructor.

DFDMap **Map*(void)

Returns the map.

VObject **DoMakeContent*(void)

Establishes the view position.

Menubar **DoMakeMenuBar*(void)

Sets up the main menu entries.

Menu **MakeMenu*(int menuId)

Links submenu entries to main entries of menu *menuId*.

void *DoSetupMenu*(Menu* m)

Establishes which menu items will be bolded and which will appear as gray initially in menu *m*.

Command **DoMenuCommand*(int)

Command defines the actions to be taken if a menu item is selected when the option is bolded.

bool *DoWrite*(OStream&s, Data *data)

Defines how information is written to the output stream if the file is saved to stream s

bool *DoRead*(IStream&s, Data *data)

Defines how data is read from an input stream s when a saved file is recalled.

void *DoMakeDiagr*(void)

From the initial information in the intermediate file with extension .tab, a collection of process objects and a collection of action line objects is assembled and temporarily held in the document. These are then used to collect together the process and line viewable objects and to pass them to the view for storage and redrawing.

void *DoRemakeDiagr*(void)

This obtains the current statuses of the processes and actions from the map. The statuses are saved and the information passed to the view so it can be redrawn to represent the most recent state of the diagram..

void *DoSetStatus*(OrdCollection* c, int j)

Matches each of the items in the OrdCollection *c* to the Collection of shapes representing the diagram: if they correspond it sets the status *j* which will be 0 if inactive, 1 if potential or 2 if active for processes and 0 for inactive, 1 for potential for action lines.

void *DoSetallStatus*(bool)

Sets all process and action statuses to inactive.

int *SetLab*(Shape *sh)

Store a label name which the user indicates by clicking on an action line with the mouse during simulation. When the user indicates the input action line the label name is stored in *lab1* when the output action line is indicated the label name is stored in *lab2*.

void *ResetLabs*(void)

Nullifies the label names saved from last simulation input.

3.2.2 DFDView Class

The class **DFDView** provides a drawing surface to display the data structures of the application. The view is empty initially and redrawn when a dataflow diagram is loaded or changed. It also contains the methods for dispatch of events such as selection of an object with the mouse which take place within the view window. Views are usually put into either a clipper or a scroller in order to display only a reasonable area of the view.

Superclass

View

Attributes

int simflag	a flag which indicates whether in simulation mode
int numproc	the number of process shapes in the diagram

Public Methods

DFDView(Document* d, Point ext)

View constructor: set up a view related to document *d* at point *ext*

~DFDView(void)

View destructor.

Command **DispatchEvents*(Point p, Token& t, Clipper* cl)

This method examines the input token t and dispatches an event which has taken place within the view to the appropriate event handler. In this case selection of a shape at point *p* with the left mouse button (defined for each of PrShape and LnShape classes) triggers an event. For mouse events such as left and middle button depression the clipper *cl* executes a mouse tracking command.

void *SetShapes*(SeqCollection *list)

Sets the shapes from the view's *list* of shapes into the view's container.

OSstream *&PrintOn*(OStream & s)

Returns data to be saved on an output stream *s*.

IStream *&ReadFrom*(IStream& s)

Reads from input stream s and redisplays view.

void *CollectParts*(Collection *c)

Incorporates the view's list of shapes into a collection *c*.

3.2.3 Classes for storage and presentation of shapes in the view

3.2.3.1 Shape Class

Defines a basic shape object which can appear in a view.

Superclass

VObject

Public Methods

Shape(Rectangle r)

Shape constructor: for a shape defined as drawn within a bounding box defined by the Rectangle r.

Metric *GetMinSize*(void)

Get minimum size.

void *Draw*(Rectangle)

Not defined.

OStream *&PrintOn*(OStream& s)

Sets output stream for save.

IStream *&ReadFrom*(IStream & s)

Sets input stream for retrieve.

3.2.3.2 Aline Class

Stores an action line characteristics as known initially from its start and end process information. Once this has been established the actual end points and arrow and label positions can be calculated.

Superclass

Object

Public attributes

Point sp	initially holds the centre point of the process at the start of the action line, later the actual start point
Point ep	holds the centre point of the process at the end of the action line
Point nsp	start point for line with allowance for process circle shape
Point a1	end point of one part of arrow
Point a2	end point of other part of arrow
byte pn1[80]	name of process at start of line
byte pn2[80]	name of process at end of line

byte lab[80]		action line label name

Public methods

aline(byte pn1[], byte pn2[], byte lab[] , Point sp, Point ep)

Aline constructor: initialize pn1, pn2, lab, sp and ep.

void *recalc*(void)

Points nsp and ep are initially the centre points of the processes which are at the extreme of the line. The method calculates nsp which is the actual line start point and recalculates ep to be the actual end point. Using the line gradient, the end points of the arrow a1 and a2 are calculated so the arrow will be positioned at the end point of the action line.

3.2.3.3 LnShape Class

This class defines the properties of an action line shape and how it should be drawn in relation to them.

Superclass

Shape

Private attributes

Point sp		start of line

Point ep		end of line

Point a1	end of arrow line
Point a2	end of arrow line
byte lab[80]	action line name
int st	draw status (zero normally: set to one when line is potential simulation action)

Methods

LnShape(Rectangle r, Point s, Point f, Point aa1, Point aa2, byte lb[], DFDView * v)

LnShape constructor: set up line characteristics within a bounding rectangle r within view v; the start point *sp* is set to s, the end point *ep* to f, arrow endpoints to aa1 and aa2 and the name *lab* to lb.

Command **DoLeftButtonDownCommand*(Point p, Token t,int cl)

Defines what action is to be taken when there is a left mouse click on an action line in the view window. If there is no *DoLeftButtonDownCommand* defined already on the View or DFDView class then this method will take effect if an LnShape object is selected. If the Point is contained in the LnShape object then the event is passed to method *DispatchEvents in the DFDView* and a user defined command (here DFDCommand) is returned for later execution by the document.

void *SetStatus*(int)

Sets the line status as defined above under attributes.

void *Draw*(Rectangle r)

Defines how the line shape is to be drawn within its bounding rectangle r and according to its current status.

3.2.3.4 DProcess Class

Holds temporary information about the processes in a diagram.

Superclass

Object

Attributes

int ttype	process type: 0 for terminator, 1 for non-terminal process
int tstatus	status: 0 is inactive, 1 is potential, 2 is active
int tlevel	process level in the diagram
Point cp	centre point of process
byte tname[80]	process name
byte tstate[80]	process state information (for future use)

Method

Process(byte tn[], int l, int t, int st, int cx, int cy)

Process constructor: save the characteristics of the process with l as *tlevel*, t as *ttype*, st as *tstatus* and cx, cy as the Point *cp*.

3.2.3.5 PrShape Class

This class defines the characteristics of the process shape object and how it is to be drawn in the view.

Superclass

VObject

Attributes

int ttype	process type as in (dprocess class)
int tstatus	status (as in dprocess)
byte tname[80]	name

Public methods

*PrShape(*Rectangle r, byte tn[], int tt, int ts, DFDView * v)

PrShape constructor: establishe the initial characteristics of the process shape within bounding rectangle r within view v; the attribute tname is set to tn, tstatus set to ts and ttype set to tt.

Command **DoLeftButtonDownCommand*(Point p, Token t, int cl)

Defines the command to be used if left mouse button is clicked on a process shape If there is no *DoLeftButtonDownCommand* defined already on the View or DFDView class then this method will take effect if a PrShape object is selected. If the Point is contained in the PrShape object then the event is passed to method *DispatchEvents* in *DFDView* and a user defined command (here RANCommand) is returned for later execution by the document.

void *Draw*(Rectangle r)

Defines how a process shape is to be drawn within its bounding rectangle r. The appearance of the process shape is dependent on whether it is a terminal or internal process and its current simulation status, inactive, active or potential.

3.2.4 Diagram Command Classes

There are two command classes associated with active simulation mode. If the user does a mouse click on a potential input line followed by a mouse click on a potential output

action line then the user has made a specific choice for the next simulation step: DFDCommand command is prompted to determine what is to be the next state of the diagram. If the user clicks on a potential process during active simulation then the user wants the system to make a random choice for the next simulation step: in this case the RANCommand is prompted.

3.2.4.1 DFDCommand Class

This command is to be enacted when both a potential input and a potential output action line has been selected by left clicking the mouse when in simulation mode. It means that the user has made a specific choice for the next simulation step.

Superclass

Command

Attributes

DFDDocument* doc	the current document
Shape *sh	the selected shape

Methods

DFDCommand(int cno, const char* cn, DFDDocument* d, Shape* s):Command(cno, cn)

DFDCommand constructor: save the document and the selected shape; the user defined command is given unique number (>= 1000) and a name using the parameters *cno* and *cn*.

void *DoIt*(void)

Defines what is to occur when the user has made a specific selection for the next simulation step by clicking on an input and an output potential action line in succession. The names of the chosen action lines are transmitted to the map so that it can convey the information to CWB and readjust its current state to the next step.

3.2.4.2 RANCommand Class

This command is to be enacted when the user has made a random selection for the choice of the next simulation step. The user does this by depressing the left mouse button on a potential (bolded) process.

Superclass

Command

Attributes

DFDDocument* doc	the current document
Shape *sh	the selected shape

Methods

RANCommand(int cno, char* cna, DFDDocument* d, Shape* s) : Command(cno, cn)

RANCommand constructor: save the document and the selected shape, the user defined command is given unique number (>= 1000) and a name using the parameters *cno* and *cn*.

void *DoIt*(void)

Defines what is to occur when the left mouse is clicked on a potential process shape during simulation mode. The map is instructed to convey a random action to CWB and to adjust its current state accordingly.

3.3 Design: CWB Interface

CWB Commands, Potential Actions and DFD Status

This section describes all the objects that provide an interface to **CWB** process. They have some knowledge about the **CWB** process. The **CWB** commands are responsible for individual commands and know about the syntax of the commands for both the input and output of their specific command. The **DFD** Status knows about the *status* information returned by **CWB** process and, understands the syntax of the **CWB** output for the status lines. The Potential Actions know the syntax of the potential lines and can extract the information from it. The **DFD** map provides the high level interface between all the objects that interface with the **CWB** process.

3.3.1 CWB Process

There is one **CWB** process associated with each **DFD** Document.

The **CWB** process keeps track of

- what the **DFD** can do next in the simulation (all potential actions)
- past transitions (actions) that have occurred

The **CWB** process can change the simulation status of the **DFD** by

- going to a *next* **DFD** status through the selection of a potential action
- going back to a *previous* **DFD** status

The objects in the **DFD** project present another *view* or representation of the information contained in **CWB** process in the form of a **DFD** in action.

3.3.1.1 CwbComm Class

CWB communication is responsible for providing the interface between the **CWB** process and the *commands* of the application.

It is responsible for starting the **CWB** process and for communicating with **CWB** process by opening a bidirectional pipe. It has the handles to the input and output pipes of the **CWB** process

It sends the commands to **CWB** process and directly receives the output from **CWB** process. It does not, however, have any knowledge about the *semantics* or *contents* of the input or output, nor when the **CWB** process has finished one command and is ready to receive another. For this reason, it does not control the receipt of the output and only receives one character at a time.

It is instantiated for each **DFD** Document.

Superclass

None

Attributes

FILE *toCwb	file handle for input *to* **CWB** process
FILE *fromCwb	file handle for output *from* **CWB** process

int cwbIn[2] Input pipe for bidirectional communication

int cwbOut[2] Output pipe for bidirectional communication

bool needFlush indicates if need to flush the input pipe *to* **CWB** process when receiving

Methods

CwbComm(void)

CwbComm constructor starts a **CWB** process and make it ready to send commands: create the bidirectional communication, fork the **CWB** process, open the input and output pipe, and flush the output pipe.

~CwbComm(void)

CwbComm destructor stops the **CWB** process by sending the *quit* command.

void *Send*(const char *command)

Sends *command* to **CWB** process on input pipe; it does not know anything about the syntax of the command.

char *Receive*(void)

Receives the **CWB** output (from **CWB** process), 1 char at a time on output pipe, it does not know anything about the syntax of the output

3.3.2 CWB Commands

This groups all the commands that affect the **CWB** process in the normal mode of operation. These commands do not affect the status of the **DFD**.

CwbCommand is the generic class and all the other **CWB** Commands inherit this class. The subclass command classes has the method *DoIt* which always include two steps: send message to **CWB** process and receive the output from **CWB** process Most commands also display some output to the user.

These commands are created by DFDMap and executed later by the CommandProcessor with the method *DoIt*.

3.3.2.1 CWBCommand Class

This is the generic **CWB** command interface and it is never instantiated directly. It is the only class that accesses **CWB** communication object directly. So, it provides an interface between all the commands and the CwbComm object. All **CWB** commands inherit this class.

It is responsible to receive the **CWB** output and put it into a format that can be used by the **DFD** application. It knows about the general format of the **CWB** commands and does some basic formatting in order to send the commands to the **CWB** process. It is responsible for the low level formatting of the commands: it knows the delimiters of the **CWB** output and parses the output by discarding end of command data.

Superclass

Command

Attributes

_cwb:	handle to CWB process (communication object) for this command
cwbLineList	a list of all output lines from CWB process *without* the discarded information

Methods

CwbCommand(CwbComm *cwb, const char *cmdName): Command(cmdName)

CwbCommand constructor saves the handle to the CwbComm object and initializes CwbLineList. The cmdName is passed to Command.

~CwbCommand(void)

CwbCommand destructor deletes CwbLineList.

void *Send*(const char *command)

void *Send*(const char *command, const char *name)

void *Send*(int command)

void *Send*(const char *command, int nb)

Send the *command* to **CWB** process using CwbComm. Since it knows about the general syntax of the commands, it is able to accept the 4 different formats of the commands with 4 different signatures.

void *ReceiveAll*(void)

Receive all of the **CWB** output for the command sent and parse it, discarding any data that is not useful for **DFD** application. The result is stored in cwbLineList

const char **GetDisplay*(void)

Return the display format of the **CWB** output that can be used with the MessageDialogs for display to the user

SeqCollection **GetList*(void)

Return the cwbLineList format of the **CWB** output. Each item in the SeqCollection represents a separate line from the **CWB** output and is used by the **CWB** commands to create the status of different objects.

void *DeleteList*(SeqCollection *l)

Deletes the contents of SeqCollection *l.

3.3.2.2 OpenFile Class

Command that sends *open file or load file* command to **CWB** process. It interprets the output from the *open file* command. It checks that the file exists. **CWB** process also validates the **CWB** file. It displays any of these errors to the user.

After successfully opening the **CWB** file, it determines the main process name of the **DFD**. It assumes that there is one main process and one **DFD** in a file.

Superclass

CwbCommand

Attributes

mainProcess	main process name of current DFD
cwbFile	File name that needs to be opened
_doc	handle to **DFD** document

Methods

OpenFile(DFDDocument *doc, CwbComm *cwb, String fileName) : CwbCommand(cwb, "OpenFile")

OpenFile constructor: save fileName in cwbFile and save doc handle in _doc; give the cwb handle and the cmdName "OpenFile" to CwbCommand.

~OpenFile(void)

OpenFile destructor: delete cwbFile name.

const char **GetMainProcess*(void)

Return the main process name in cwbFile.

void *CloseFile*(void)

Send the *close file* command to **CWB** process.

void *DoIt*(void)

Check validity of cwbFile; if it is not valid, warn user and return

Get the main process from cwbFile.

Tell **DFD**Map the main process name (through GetMainProcess)

Display the **DFD** diagram.

Send the *close file* command so that there is no conflict of data in **CWB** process (with *CloseFile*).

Send the *open file* command to **CWB** process.

3.3.2.3 Deadlock Class

Command that sends the *deadlock* command to the **CWB** process. It interprets the output from *deadlock* command and, depending on the output, displays the results to the user

"No deadlock" is displayed if there are none. If there are some deadlock states, then they are displayed to user in the CWB output format.

Superclass

CwbCommand

Attributes

deadlockProcess	deadlock condition is tested for the deadlockProcess

Methods

Deadlock(CwbComm *cwb, const char *processName):CwbCommand(cwb, "Deadlock")

Deadlock constructor saves the deadlock processName; it gives the cwb handle and the cmdName "Deadlock" to CwbCommand.

~*Deadlock*(void)

Deadlock destructor.

void *DoIt*(void)

Send *deadlock* command to **CWB** process and receive the **CWB** response by using CwbCommand. Interpret the output and display the result to the user.

3.3.2.4 Size Class

Command that sends the *size* command to **CWB** process. It interprets the output from the *size* command and displays the result to the user: the **CWB** output from the command

Superclass

CwbCommand

Attributes

sizeProcess	size is determined for the sizeProcess

Methods

Size(CwbComm *cwb, const char *processName) : CwbCommand(cwb, "Size")

Size constructor saves the size processName; it gives the cwb handle and the cmdName "Size" to CwbCommand.

~*Size*(void)

Size destructor.

void *DoIt*(void)

Send *size* command to **CWB** process and receive **CWB** response by using CwbCommand Interpret the output and display the result to the user

3.3.2.5 Equivalence Class

Command that sends the *equivalence* command. It interprets the output from the *equivalence* command and displays the result to the user. If the two processes are not equivalent, it also displays the weak modality HML formula distinguishing between the two processes {M 92, p. 19].

Superclass

CwbCommand

Attributes

process1	1st process name needed for equivalence test
process2	2nd process name needed for equivalence test

Methods

Equivalence(CwbComm *cwb, const char *process1Name, const char *process2Name) :
CwbCommand(cwb, "Equivalence")

Equivalence constructor saves the 2 process names for equivalence test: process1 and process2; it gives the cwb handle and the cmdName "Equivalence" to CwbCommand.

~*Equivalence*(void)

Equivalence destructor.

void *DoIt*(void)

Send *equivalence* command to **CWB** process and receive **CWB** response by using CwbCommand. Interpret the output and display the result to the user.

3.3.3 *SimCommands*

This groups all the commands that affect the simulation or dynamic action of the **DFD** These commands can either modify or make queries about the *current status* of the **DFD** These commands are only valid after the simulation has started (with exception of *Start* simulation).

3.3.3.1 SimCommand Class

This is the generic Simulation command interface and it is never instantiated

It is responsible with determining all the dynamic information about the DFD and provides *UpdateDFD* for this purpose. This method is called by all the simulation commands to update the current **DFD** status and potential action data.

All the simulation commands are subclasses of SimCommand which, in turn, is a subclass of CwbCommand. SimCommand uses CwbCommand to interface with **CWB** process SimCommands are created by the **DFD** Map

Superclass

CwbCommand

Attributes

_doc handle to the **DFD** document that wants to do SimCommand

Methods

SimCommand(DFDDocument *doc, CwbComm *cwb, const char *cmdName) : CwbCommand(cwb, cmdName)

SimCommand constructor: save doc handle in _doc; give the cwb handle and cmdName to CwbCommand.

~*SimCommand*(void)

SimCommand destructor

void *UpdateDFD*(void)

Update the current **DFD** status and potential actions in the map.

Redisplay the **DFD** diagram.

3.3.3.2 Start Class

This starts the simulation by sending *start simulation* command to **CWB** process.

Superclass

SimCommand

Attributes

simProcess	Start simulation on simProcess from the **DFD**

Methods

Start(DFDDocument *doc, CwbComm *cwb, const char *processName)

SimCommand(doc, cwb, "Start")

Start constructor: save start processName in simProcess; give doc and cwb handle, and cmdName "Start" to SimCommand.

~*Start*(void)

Start destructor.

void *DoIt*(void)

Send *start simulation* command to **CWB** process and receive **CWB** response by using CwbCommand. Update the **DFD** map status and **DFD** diagram

3.3.3.3 Stop Class

This stops the simulation by sending *stop simulation* command to **CWB** process

Superclass

SimCommand

Attributes

None

Methods

Stop(DFDDocument *doc, CwbComm *cwb) : SimCommand(doc, cwb, "Stop")

Stop constructor: give doc and cwb handle, and cmdName *stop* to SimCommand.

~*Stop*(void)

Stop destructor.

void *DoIt*(void)

Send *stop simulation* command to **CWB** process and receive **CWB** response by using CwbCommand. Update the **DFD** map status (back to default values) and **DFD** diagram.

3.3.3.4 Next Class

This does the next simulation transition by sending *next* command to **CWB** process.

Superclass

SimCommand

Attributes

nextStep	next transition step from itemNb in potential action

Methods

Next(DFDDocument *doc, CwbComm *cwb, int nbSteps) : SimCommand(doc, cwb, "Next")

Next constructor: save nbSteps in nextStep; give doc and cwb handle, and cmdName "Next" to SimCommand.

~Next(void)

Next destructor.

void *DoIt*(void)

Send *next action* command to CWB process and receive CWB response by using CwbCommand. Update the DFD map status and DFD diagram.

3.3.3.5 Random Class

This makes a number of simulation transition steps in the DFD by randomly selecting the number of transition steps.

Superclass

SimCommand

Attributes

nbSteps number of random transition steps to make

Methods

Random(DFDDocument *doc, CwbComm *cwb, int numberSteps) : SimCommand(doc, cwb, "Random")

Random constructor: save numberSteps in nbSteps; give doc and cwb handle, and cmdName "Random" to SimCommand.

~*Random*(void)

Random destructor.

void *DoIt*(void)

Send *random* command for *number steps* to **CWB** process and receive **CWB** response by using CwbCommand. Update the **DFD** map status and **DFD** diagram.

3.3.3.6 Previous Class

Returns to a previous **DFD** status which occurred within the simulation session. This command is triggered through the menu item.

Superclass

SimCommand

Attributes

None

Methods

Previous(DFDDocument *doc, CwbComm *cwb) SimCommand(doc, cwb, "Previous")

Previous constructor: give doc and cwb handle, and cmdName "Previous" to SimCommand.

~*Previous*(void)

Previous destructor.

void *DoIt*(void)

Send *history* command to **CWB** process and receive **CWB** response by using CwbCommand. This gets all transitions made in the simulation session Display the transitions to allow the user to select a previous **DFD** status.

If the user selects a previous **DFD** status, send *return* command to return to a previous **DFD** status. Update the **DFD** map status and **DFD** diagram Otherwise, don't do anything

3.3.4 DFD Status Classes

These classes are used to represent the status of the **DFD**. This includes the status of each process in the **DFD** and the potential actions associated with a **DFD** in a particular status

3.3.4.1 CwbLine Class

It holds the ET format of the CWB output that can be displayed and makes it possible to select any items in select lists.

Superclass

TextItem

Attributes

String line	string format of **CWB** output line
int index	index number of line (needed when *select* the item)

Methods

CwbLine(String *l, int ind)

CwbLine constructor: save l in line and ind in index; give line to TextItem.

~*CwbLine*(void)

CwbLine destructor.

void *SetString*(const char *str)

Save str in line; and gives str to TextItem.

const char **GetString*(void)

Return const char * of line

int *GetIndex*(void)

Return index of CwbLine.

3.3.4.2 ProcessStatus Class

The **DFD** Process status represents the status of a single process in the **DFD.**

It can be either a *current* or *potential* **DFD** Process status.

It is responsible for parsing the **CWB** process item, which is the part of the **CWB** process status between the vertical bars. It knows how to parse the process information from: (... | ['input1.]['input2.][output1.][output2.]process name | ...).

A process status always has a process name but not necessarily any activated inputs or activated outputs. If there is no input or output, it means that the process is idle (not activated). If there is at least one activated input, then the process is activated.

Superclass

CwbLine

Attributes

SeqCollection *inputActions	list of *activated* input actions
SeqCollection *outputActions	list of *activated* output actions
String processName	name of the process
String processItem	string format of cwbProcessItem

Methods

ProcessStatus(String cwbProcessItem)

ProcessStatus constructor: create the ProcessStatus by parsing cwbProcessItem and creating the list of activated input actions and activated output actions; save cwbProcessItem in processItem, extract the process name from it and save it in processName.

~*ProcessStatus*(void)

ProcessStatus destructor: destroy the list of activated input and output actions.

const char **GetProcessName*(void)

Return the name of the process.

SeqCollection **GetInput*(void)

Return list of activated inputs.

SeqCollection **GetOutput*(void)

Return list of activated outputs.

bool *IsActivated*(void)

Return TRUE if the process is activated. Else return FALSE (process is idle).

bool ==(ProcessStatus &s)

Return TRUE if s is equal to this process status using processItem. Else return FALSE

3.3.4.3 DFDStatus Class

This is a collection of process status. It is responsible for doing the high level parsing of the CWB output line and for creating the process status by delimiting the CWB process items.

It can be either a *current* or *potential* **DFD** status.

Superclass

CwbLine

Attributes

SeqCollection *processStatusList	list of all the process status
String dfdStatus	string format of the **DFD** Status

Methods

DFDStatus(String line)

DFDStatus constructor: parse the cwb line output by determining where the cwb process items are and create processStatus for each item.

~DFDStatus(void)

DFDStatus destructor: destroy each processStatus.

String *GetDFDStatus*(void)

Return String format of DFDStatus for display.

ProcessStatus *GetProcessStatus*(const char *processName)

Return ProcessStatus for processName in the **DFD**Status; if processName is not valid, then return NULL.

SeqCollection **GetAllProcessStatus*(void)

Return list of all the ProcessStatus in **DFD**Status.

bool *IsActiveStatus*(void)

Return TRUE if any of the processes in **DFD**Status is activated. Else return FALSE.

3.3.4.4 PotentialAction Class

It represents a single potential action with all the information required to represent it on the **DFD** diagram (the input action, the process, and the output action) and to use as a command with **CWB** process (potential action number).

There are 2 different mechanisms used to get this information.

1. It uses the potential action line to determine the input action and number (determined by *n*: ---- *i* ----> (...); where *n* is potential action number and *i* is the input action)

2. It uses both the current and the potential DFD status to determine the output action and potential action process name. This is set by the DFD map since it has knowledge about the current DFD status.

Superclass

CwbLine

Attributes

int number	potential action number
String *inputAction	input action that makes the potential action transition
String *outputAction	output action that is triggered if the potential action is selected
String *processName	process that will be affected if the potential action is selected
DFDStatus *dfdStatus	potential (or resulting) DFD status if potential action is selected

Methods

PotentialAction(String cwbLine)

PotentialAction constructor: parse the cwbLine and save the number and inputAction; create the potential dfdStatus.

~*PotentialAction*(void)

PotentialAction destructor: destroy processName, inputAction, outputAction and dfdStatus.

int *GetNumber*(void)

Return the potential action number.

String **GetInputAction*(void)

Return the input action that will would make the potential action transition if selected.

String **GetOutputAction*(void)

Return the output action affected by the potential action.

const char **GetProcessName*(void)

Return the process name affected by the potential action.

DFDStatus **GetDFDStatus*(void)

Return the *potential* dfdStatus. This is the resulting status of the DFD if the potential action is selected in the simulation.

void *SetOutputAction*(String *outAct)

Save outAct in outputAction.

void *SetProcessName*(const char *name)

Save name in processName.

3.3.5 *Map between Application and CWB*

The DFDMap is the only class in this section [3.1 Design: Interface with **CWB**] that is accessed directly by the rest of the application. It provides the link, or the map, between the DFDDocument, the **DFD** diagram, the **CWB** commands and the **DFD** status

There is one DFDMap for each DFDDocument. It is responsible for starting the **CWB** process associated with the DFDDocument.

The DFDMap is driven by both menu input and from mouse input to the **DFD** diagram.

3.3.5.1 DFD Map Class

It keeps track of the status of the **DFD** diagram and the **CWB** process. It starts the **CWB** process for the current DFDDocument. It knows the status of each process in the **DFD** diagram, and of each potential action. It knows whether the **CWB** process has loaded a **DFD** diagram and whether it is in normal or simulation mode.

It creates all the **CWB** commands (DoMake methods), for query and modify, in both normal and simulation mode.

It is also responsible for updating the **DFD** status and for creating the potential actions. It does this when told by a simulation command that the status has been modified

It also provides the interface to *query* about the status of **DFD** (current process and **DFD** status) and the potential actions. It provides the interface to modify status of the **DFD** by setting any simulation input/output pair.

It stores all the information required by the application for DFD concerning the CWB process or the status of the DFD.

Superclass

None

Attributes

DFDDocument *_doc	handle to DFDDocument (needed to update the DFDDiagram)
SeqCollection *potActList	list of all the potential actions
DFDStatus *dfdStatus	current DFD status
CwbComm *cwb	handle to active CWB process
String display	CWB output with no formatting.
String mainProcess	main process in the DFD
bool simStartStatus	boolean indicating whether simulation is started/not started
bool fileOpenStatus	boolean indicating whether a CWB file is open/not opened

Methods

DFDMap(DFDDocument *doc)

DFDMap constructor: start the CWB process and save it's CwbComm handle in cwb; save doc handle in _doc.

~DFDMap(void)

DFDMap destructor: stop CWB process by destroying cwb; destroy potActList and dfdStatus.

void *SetMainProcess*(const char *processName)

Save processName in mainProcess; this is set by FileOpen.

const char **GetMainProcess*(void)

Return mainProcess name.

const char **GetDisplay*(void)

Return display; set by the last CWB command that did *DoIt*( ).

void *SetFileOpen*(bool openFile)

Set openFileStatus to openFile; set by OpenFile.

bool *IsFileOpen*(void)

Return openFileStatus.

Do Make CWB commands: All of these commands are made in the DFDMap and executed by CommandProcessor.

OpenFile **DoMakeOpenFile*(String *filename)

Make and return *OpenFile* command for filename.

Size **DoMakeSize*(void)

Make and return *Size* command for mainProcess.

Deadlock **DoMakeDeadlock*(void)

Make and return *Deadlock* command for mainProcess.

Equivalence **DoMakeEquivalence*(String *eqprocess)

Make and return *Equivalence* command for mainProcess and eqprocess.

Do Make CWB simulation commands: All of these commands are made in the DFDMap and executed by CommandProcessor.

Start **DoMakeStart*(void)

Make and return *Start* command for mainProcess.

Next **DoMakeNext*(int stepNb)

Make and return *Next* command for mainProcess.

Next **DoMakeNext*(void)

Make and return *Next* command for mainProcess.

Random **DoMakeRandom*(int nbSteps)

Make and return *Random* command for mainProcess.

void *DoMakeRandom*(void)

Make and return *Random* command for mainProcess.

Previous **DoMakePrevious*(void)

Make and return *Previous* command for mainProcess.

Stop **DoMakeStop*(void)

Make and return *Stop* command for mainProcess.

DFD status methods: All of these methods either create or return the status of the **DFD** The status is only modified while in simulation mode of operation.

void *Update*(const char *cmdDisplay)

Update the **DFD** status and create the new potential actions. Each **CWB** command that modifies the **DFD** status calls *Update*( ). Save the cmdDisplay in display.

SeqCollection **GetPotentialActions*(void)

Return the potActList.

DFDStatus **GetCurrentDFDStatus*(void)

Return the currentDfdStatus; the methods for the process or dfd status are used to get details of the **DFD**.

PotentialAction **GetSimInOutPair*(const char *inputAction, const char *outputAction)

Returns the PotentialAction that has the inputAction/outputAction pair. Returns NULL if none exist.

void *GetActiveProcess*(SeqCollection *seqColl)

Create the list seqColl of processes that are activated for the current **DFD** status.

void *GetPotProcess*(SeqCollection *seqColl)

Create the list seqColl of processes that are part of a potential action for the current **DFD** status.

void *GetPotOut*(SeqCollection *seqColl)

Create the list seqColl of output actions that are part of a potential action for the current **DFD** status.

void *GetPotIn*(SeqCollection *seqColl)

Create the list seqColl of output actions that are part of a potential action for the current **DFD** status.

Simulation methods: these methods are used by **DFD** diagram while in simulation mode to query about the mode of operation or to select a potential action.

bool *IsSimStart*(void)

Return simStartedStatus.

void *SetSimStart*(bool simStart)

Set simStartedStatus to simStart.

bool *IsSimInOutPair*(const char *inputAction, const char *outputAction)

Return TRUE if inputAction/outputAction is a valid input/output simulation pair for the current **DFD** status.

bool *SetSimInOutPair*(const char *inputAction, const char *outputAction)

Set the inputAction/outputAction simulation pair by determining which potential action the pair belongs to and by sending the next command for that potential action.

4 Conclusions

It has been shown that it is theoretically possible to simulate the behavior of a data flow diagram using the CWB tool in [BGST]. In this project, we have shown that it is also possible to do it in practical terms.

The project has successfully accomplished the following:

- We have provided a graphical interface that represents a DFD in CCS and simulate its behavior using CWB tool.
- With the Π-DFD Graphic Interface, it is possible to easily analyze and simulate the behavior of a DFD defined in tuple format.
- Defined an easy mechanism for the positioning of the objects in the data flow diagram.
- We have successfully incorporated the system into ET++ framework.

Π-DFD Graphic Interface provides a practical system that makes it relatively easy to test the usefulness and applicability of different commands available in the CWB tool.

We have provided flexibility in the design of the system to allow for future enhancements and have separated the responsibilities of the objects in the system: display, positioning, semantics (status information), commands, and the CWB tool.

The CWB tool is an integral part of the P-DFD Graphic Interface, and the following was accomplished for the simulation and analysis of the data flow diagram:

- successful extraction of information from the CWB output enabling simulation of the *behavior* of a data flow diagram.
- selection of a small subset of commands from the CWB tool that are useful for the *analysis* of a data flow diagram.

The CWB Interface subsystem does the following:

- provides the interface to the CWB tool.
- extracts all the useful information from the CWB for potential actions and status of the processes and actions in the data flow diagram.
- provides an output format that is independent of the CWB format.
- is consistent so that modifications are simplified.
- provides families of objects through inheritance: normal mode CwbCommands, simulation mode SimCommands, DFDStatus that include process statuses, and potential actions.

There are many improvements that can be made to the system and these are listed in the next section.

4.1 Future Improvements

Improvements can be made by both taking advantage of the functionality available within the ET++ framework, by providing a better user interface for the DFD analysis provided, and by exploiting the functionality available in CWB tool, and by allowing more flexibility

in the analysis and display of the data flow diagram. Here is a description of some of the more important ones.

4.1.1 Provide Editing capabilities

For this implementation, the user has to start with a paper copy of a data flow diagram which is scanned in, features are extracted, then there is a syntactic analysis and semantic analysis. This results in the semantic structure which is defined in CWB code. This is required for legacy systems but it is an unreasonable requirement for a new design.

A significant improvement would be the provision of editing capabilities on the data flow diagram. An improved approach would be to use Π-DFD Graphic Interface for *creating* the data flow diagram and generating the logical structure from it. This would make it possible for the software designer to modify the data flow diagram. This is feasible within the ET++ framework environment; ET++ provides the capability for easily storing the *view* of the data flow diagram: the graphical and text entities used to display the DFD, and the positions of each entity. This would be stored in the tuple format and display in the same manner as it is presently done in this implementation.

ET++ does not provide the application-specific *semantic* checking for the tuple format generated from the display of the data flow diagram. In order to validate the semantic definition of DFD, it would be necessary to use the same logic that is required to validate the syntax and semantics of the tuple format of the data flow diagram from paper format.

An example of semantic checking is ensuring that each process has at least one input action and one output action. This semantic checking should be done when the user saves the DFD (with a warning if the semantic checking is not valid), and with a menu item that checks semantic validity.

This area would be extremely useful addition to the system since it is now required to manually place the processes by defining the coordinates.

4.1.2 Provide Undo/Redo capabilities

To improve the user interface, it would be important to provide the undo/redo capabilities for the commands implemented in the Π-DFD Graphic Interface. It is especially important for the simulation commands since they affect the status of the DFD.

To implement the undo, additional information is required and the previous DFD status should be retained. Then it would be necessary to update the DFD display *and* CWB to the previous DFD status. To put CWB status back to previous DFD status with the same mechanism that is used for Select Previous. For a random command, it is necessary to get CWB to the same DFD status while giving the same output for Select Previous To redo a command, it is necessary to retain the potential action that was selected.

For this implementation, all commands are executed by the command processor However, it is not necessary for Size, Deadlock and Equivalence commands to inherit Command since their undo/redo methods would not do anything

4.1.3 Improve User Interface for Deadlock States

For this implementation, the display of the deadlock state is simply the output from CWB output or a series of DFDStatus. This is only useful for a user who understands CWB output.

Each deadlock state is actually the DFD with activated processes. In the simulation mode of operation, the activated and potential processes are displayed. So, each deadlock is attainable in the simulation mode and should be displayed in the same manner it would be in simulation. The same mechanism that is used for the simulation mode can be used i.e. it is simply the display of the DFD by checking the status of each process and action in the DFD. Alternatively, each action taken to attain the deadlock state could be displayed in a time interval.

The list of deadlock states could be displayed in a selectable list and each deadlock state could be displayed in a new window.

The modifications required would be to allow the user to select a deadlock state, update the DFDMap status for the selected deadlock state, and make the data flow diagram in a new window for this deadlock state. This would provide visual display of the DFD in a deadlock state that would be useful to someone who does not understand the CWB output.

4.1.4 Making better use of CWB for Equivalence

The equivalence checking is currently implemented by checking two DFDs defined in the same file. This is necessary since CWB uses the following logic to define processes (or agents): if the same process name is used, then the second definition overwrites the first. For this reason, we have decided to only allow one file to be opened at a time. This means that it is necessary to include two DFDs in the same file and ensure that the process names in each DFD are unique.

A solution to this would require a modification that would map each process name in the tuple format to a displayName and a cwbName. The data flow diagram display would label each process with the original tuple name but the CWB code would have a unique name for each process.

4.1.5 More flexibility: selection of Process Name

For this implementation, the main process name is always used for all the analysis of the complete DFD. CWB tool and all the low level CwbCommands and SimCommands can accept any process name: States, Equivalence, and Simulation. The DFDMap depends entirely on CWB output and thus generates the status correctly regardless of the portion of the DFD are interested in.

Modifications would have to be made for the display portion of the DFD since it currently only supports the full display of the DFD

Partial simulation of the DFD would be useful especially if it is large and only a portion of it is to be analyzed.

4.1.6 Save Simulation DFD

The data flow diagram is activated after selecting several potential actions while in the simulation mode. Saving this activated data flow diagram would be a very useful feature for the Π-DFD Graphic Interface.

This feature can be implemented because CWB is a very flexible tool that accepts an *activated* DFD as the agent for the StartSimulation command. For example, assuming that the DFD has the following CCS representation while in simulation (E0|'s.E1|E2|E3)\{r,s,u,v}. If the user wants to save the current activated DFD, the *display* format of current DFD status could be saved and simulation would start with the following CWB command: `sim (E0|'s.E1|E2|E3)\{r,s,u,v}`

4.1.7 Allow DFD with multiple levels

The current implementation only accepts a DFD that is defined as a flat DFD i.e. with all the lowest levels of the DFD. This is explained in Section 2.10.1.6.

CWB does not understand multiple levels. It takes *all* the information it is given and uses all the processes defined. Therefore, if the refinement of a process is included in the *.CWB file, then CWB will make use of it when analyzing the DFD. In order to define different levels of the DFD, the translator would have to be modified to create different

CWB files for each level of the DFD. When the user wants to look at a new level of the DFD, CWB has to load a new *.CWB file.

Loading an *idle* DFD would not be a complex task for the system (assuming that the *.CWB files are available). For example, before going into simulation, the user selects level 2 of the DFD.

For a good user interface, the user would often want to be able to start the simulation at a higher level, and somewhere in the simulation, the user wants to look in detail at the behavior of a process. For implementation purposes this is not a trivial task. The transitions that have occurred in the simulation have to be saved, the CWB code that includes the details of the current process has to be loaded, CWB *process* has to restart simulation and go thru all the saved transitions.

4.2 Power of ET++

ET++ provides a rich set of tools that can be used to display documents. We have succeeded in making use of a small subset of the items available.

The amount of code that has to be written to display the document is reduced significantly and all of the control mechanism is taken care of by ET++.

It is a well designed application framework that works very well. The problem lies more in determining how to create the application-specific behavior

4.2.1 ET++ Documentation

The documentation available for ET++ includes a description of the classes available and the general philosophy used to create ET++. The event control mechanisms are well designed as is the general architecture which includes the division of the different models of a document.

Due to the complexity of the environment, there is a longer learning curve when working with a framework, as compared to traditional library of tools. In some ways, the problem is similar to attempting to modify an existing software system although the application framework is extremely well designed. There is a whole wealth of functionality available, but it is difficult to determine what is available and especially how to make use of what is available. This means that code re-use is more difficult.

However, once a programmer becomes experienced with ET++, it is easy to realize the power and speed with which a document application can be created.

4.2.2 Cookbook approach

Although class descriptions are available for ET++, it would have been useful to have a cookbook approach for the novice ET++ programmer.

Some short examples showing only how to add a menu, example showing how to use different dialog types, explanations of where to intercept which menu item, explanations of how to display different graphical entities. This allows a user to start creating applications. This functionality would make the development and re-use much quicker

and would be very useful for the programmer. This would also be similar to a tutorial on the application framework.

Although it is a framework application, using a code re-use approach with examples is still a pretty good way to program. Stepping the programmer through the different classes available by defining: how to put text in document, how to put diagram in document, how to create a menu, how to create dialog, which class should intercept which type of menu items, how to define the menu items.

I do not feel it is necessary, especially when learning a new application framework, to understand how everything works but rather how to make things work.

4.3 Summary

We have successfully implemented Π-DFD Graphic Interface within ET++ application framework for the understanding of the behavior of a DFD in legacy systems.

It makes use of a formal representation of DFD by taking advantage of the semantics of CCS. It uses CWB tool transparently and allows for interactive and graphical *analysis* of the behavior of the data flow diagram.

There are many improvements that can be included in the system. The input to the system is paper or tuple format of the DFD. An important improvement would be the addition of editing capabilities.

We have successfully developed our system within ET++ framework and it is a very powerful tool that provides a great wealth of functionality for displaying and viewing a

document. However, the inclusion of a cookbook documentation of the framework environment would allow a programmer to take advantage of the objects provided with ET++ and to use the framework according to its design philosophy

4.4 Personal Notes

One goal I had when I started this project was to feel more confident in the windowing environment and in object-oriented design and programming using OMT and C++. I feel that I have successfully done this. It provided good practical experience in applying software engineering techniques including specification, design, development and testing while working as part of a team. In the implementation, I made good use of inheritance by defining separate responsibilities for each group of objects.

I also learned how it is possible to take a tool that was developed for one purpose, look at it in a different way, and re-use it for another purpose. The CWB tool is designed for the manipulation and analysis of concurrent systems defined in CCS whereas, in the Π-DFD Graphic Interface, it is used to simulate and analyze the states of a data flow diagram.

This project also incorporated both theoretical and practical aspects. After learning how it is theoretically possible to completely define a data flow diagram in CCS by analyzing the mathematical proofs in [BGST], we made use of this to simulate the behavior of a data flow diagram by using the CWB tool.

It was a pleasure to see, and understand, how the output from CWB tool could be applied to a DFD defined in CCS, although it was intended for the analysis of concurrent systems.

5 References

[B 95] G. Butler. *The ET++3.0 Application Framework: A Tutorial with Examples.* Department of Computer Science, Concordia University, Montreal, Quebec, Canada.

[BGST 95A] G. Butler, P. Grogono, R. Shinghal, and I. Tjandra. *Analyzing the Logical Structure of Data Flow Diagrams in Software Documents.* **Proceedings of the Third International Conference on Document Analysis and Understanding.** Montreal, August 14--16, 1995, IEEE Press, pp. 575-578.

[BGST 95B] G. Butler, P. Grogono, R. Shinghal, and I. Tjandra. *Knowledge and the Recognition and Understanding of Software Documents.* Department of Computer Science, Concordia University, Montreal, February 1995, 47 pages.

[BGST 95C] G. Butler, P. Grogono, R. Shinghal, and I. Tjandra. *Retrieving Information from Data Flow Diagrams.* **Proceedings of Second Working Conference on Reverse Engineering.** Toronto, July 14--16, 1995, Linda Wills, Philip Newcomb, Elliot Chikofsky (eds), IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 22-29.

[M 92] Faron Moller. *The Edinburgh Concurrency Workbench (Version 6.1).* Department of Computer Science, University of Edinburgh, October 1992.

[RBPEL 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. **Object-Oriented Modeling and Design.** Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[WGM 88] Andre Weinand, Erich Gamma, Rudolf Marty. *ET++ - An Object-Oriented Application Framework in C++.* Institute for Informatics, University of Zurich, Winterthurerstr. 190, CH-8057 Zurich, Switzerland, OOPSLA '88 Proceedings.

[WGM 89] Andre Weinand, Erich Gamma, Rudolf Marty. *Design and implementation of ET++, a seamless object-oriented application framework.* **Structured Programming**. 1989. pp. 63-87.

[WG 94] Andre Weinand, Erich Gamma. *A Portable, Homogenous Class Library and Application Framework.* Computer Science Research at UBILAB, Strategy and Projects; Proceedings of the UBILAB '94 Conference, Zurich, September 1994. University of Konstanz, Konstanz, 1994, pp. 66-92.

[W 90] M. Woodman. *Yourdon Dataflow Diagrams.* **The Software lifecycle.** D. Ince, D. Andrews. Cambridge 1990. pp. 129-167.