Valerie Large

The Π-DFD Graphic Interface

A Major Report

in

The Department

of

Computer Science

Presented in partial fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

April 1996

ISBN 0-612-10869-4

ABSTRACT

The Π-DFD Graphic Interface

Valerie Large

Dataflow diagrams have been a useful tool in legacy systems for describing certain functionality of software systems and are still in use today. With current programming methodologies and improved hardware there is great motivation to re-engineer legacy systems: to do this effectively the systems must be well understood.

The Π-DFD project at Concordia University was set up to harness the power of the Edinburgh Concurrency Workbench (CWB) in analyzing these dataflow diagrams. A tuple representation was derived for representing the diagrams textually and a translator written to convert the tuple format to one suitable for input to CWB. The textual interface was not very user friendly so the work of this project was to develop a graphic interface for the Π-DFD system. The ET++ application framework for C++ was used to develop this interface which enables the user to analyze dataflow diagrams and to run simulations on them. The user communicates with the system by menus and by using the mouse directly on the diagram to control simulations. The graphic simulation is displayed dynamically as it changes the status of the diagram agents.

This report describes some of the theoretical background behind the diagrams and CWB as well as the detailed design of the interface with the user and CWB. Finally it analyzes the success of the interface in general terms and specifically as an improved means of access to CWB.

ACKNOWLEDGEMENTS

I would like to thank the staff members of the Π-DFD project (Professor Butler, Professor Grogono, Professor Shinghal and I. Tjandra) for inviting me to participate on it to develop the graphic interface Professor Butler spent much time introducing us to the internal workings of the application framework ET and assisting with some of the development problems. He was readily available for any discussion on the project and I am most grateful for the time he made available to me Ono Tjandra spent time to introduce us to much of the background theory behind the Edinburgh Concurrency Workbench and focused us on particular areas of interest. I would also like to thank both Professors Butler and Grogono and the Department of Computer Science for examining this report

Finally I would like to acknowledge my husband Andy and daughters Amanda and Kirsty for all the times I was missing on evenings and weekends working in the computer laboratory and who have always supported me in my academic endeavors

Table of Contents

Table of figures

1. INTRODUCTION.

1.1 Dataflow diagrams and their context in structured design.

1.1.1 The syntax and semantics of dataflow diagrams

Dataflow Diagrams (DFDs) are a widely used notation in Software Engineering as a tool for specifying the functional aspect of a system. They describe systems as collections of data that are manipulated by functions or processes. Data in the diagrams requires several means of representation. it may be stored in data repositories, it may flow in data flows and be transferred to and from an external environment.

The DFD as a software tool has the benefit of simplicity: it can be learned easily and be understood easily It provides an easy, graphic means of modeling the flow of data through a system. Four fundamental elements of a DFD are:

1. Symbols which represent functions or processes
2. Symbols (usually arrows) which represent dataflows. Normally arrows entering processes represent input values that belong to the domain of that function: outgoing arrows represent the results of that function.
3. Symbols to represent datastores which are permanent data repositories.
4. Symbols for sources and sinks which represent data acquisition and production during computer-human interaction.

Several notations have been devised for representing these fundamental syntactical concepts. (For example, Yourdon [WOOD90], DeMarco [DEMA79] and Rumbaugh

[RUMB91]) Further rules are required to specify overall system structure, to represent decomposed diagrams, the balance of flows and the uniqueness of labeling of elements

Some of the semantics of a DFD are expressed in the choice of label names for processes and for dataflows : these normally are chosen to be meaningful but are of necessity somewhat cryptic so that data dictionaries are required to explain the semantics Because the semantics of a system are not fully expressed in the DFD it is said to be an *informal* notation.

1.1.2 The inadequacy of dataflow diagrams in isolation.

There is an innate inadequacy in using a dataflow diagram in isolation.

> *in its simplicity there is a lack of power of expression* [WOOD90,p 129]

Just as the semantics of DFDs are not fully expressed by the notation, the representation of an entire software system cannot be expressed by a DFD alone

Several means have been used to try to overcome this. Yourdon developed an extended notation for augmenting the DFDs by the addition of control flow arrows Rumbaugh[RUMB91] and Yourdon & Constantine[YOCO79] suggested using the DFD as a representation of the functional model of a system with other charts such as statecharts to represent the dynamic aspects of a system, the overall concept being regarded a *structured design* in which several modeling views are taken in order to fully represent the system. The *Statemate* Computer Aided Software Engineering (CASE) tool [ILOG91 and ILOG92] provides three views of a system the functional view represented by DFDs, a dynamic view represented by statecharts and an organizational view

represented by module-charts and in addition supplies a simulation facility to demonstrate the behavioral view

Another approach to combining the syntax and semantics is to make the notation fully formal such as Temporal CCS, the input language for the Edinburgh Concurrency Workbench (CWB) [MOLL92]. This is discussed in further detail in section 1.2.

1.1.3 Dataflow diagrams in large applications

The expression of a large system as a dataflow diagram could make the diagram so large as to make it unmanageable. to make it readable a concept of levels of decomposition can be introduced. A hierarchical system of successively refined DFDs can make a large DFD more manageable [WOOD90 and BUTL95a]. For this, the sources and sinks are always at level zero and each process may be shown in a decomposed form at a lower level: a variety of different decompositions can be shown.

1.2 The Edinburgh Concurrency Workbench

The Edinburgh Concurrency Workbench (CWB) is an automated tool used for the manipulation and analysis of concurrent systems. It can be used for equivalence and model checking using different process semantics. It is able to define behavior using a formal syntax called the *Temporal Calculus of Communicating Systems* (Temporal CCS) and can analyze the state space of a process or check semantic equivalencies. The system is used interactively with textual input and output. Once the user has created a text file in the CCS format to describe a DFD, the file can be loaded, the state spaces examined, equivalence and congruence between processes checked and simulations run on them. The simulation facility is quite sophisticated allowing the user to step through the DFD or one of the individual processes within it. The simulation steps can be individually controlled by the

user or randomly chosen by the system, breakpoints can be set and unset, a return made to any point in the simulation process and a complete simulation history given. [MOLL92], [BUTL95a]

The simple dataflow diagram ,

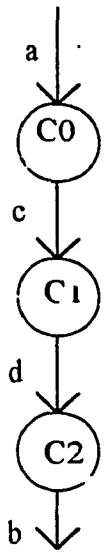


figure 1.2.1

simple dataflow diagram

can be represented by the SCCS notation:

Buff3= (C0 | C1 | C2) \{ c,d,}

C0= a.'c.C0

C1= c.'d.C1

C2= d.'b.C2

which shows three processes C0, C1 and C2 with *a* as input and *c* as output of C0, *c* as input of C1 and *d* as output and *d* as input of C2 and b as output. The simple processes C0, C1 and C2 are the processes which constitute the complex process Buff3 and c and d

dataflows are considered to be internal to the process Buff3 In CWB there is dynamic binding whenever a process description is loaded so that when comparing processes care must be taken to ensure unique process names

For a DFD where there is a process with alternative outputs, for example:

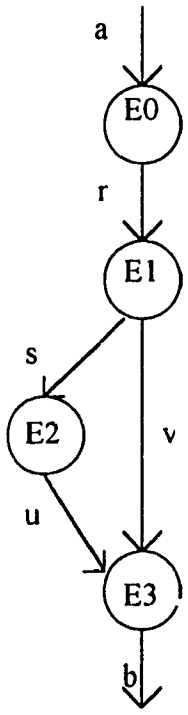


figure 1.2.2

a process with alternative outputs

the CCS input format for this partial process is

E0 = a.'r.E0

E1 = r.'s.E1 + r.'v.E1

E2 = s.'u.E2

E3 = u.v.'bE3 + v.u.'bE3

P1 = (E0 | E1 | E2 | E3) \ {r,s,u,v}

where at E1, s and v are alternatives for the output from E1. At E3 both u and v are required for the process E3 to procced to output b.

It can be seen that the formal CCS notation is a powerful means of expressing both the static and dynamic aspect of a process

For simulation the command sequence looks as follows.

Command: sim

Agent: P1

Simulated agent: P1

Transitions:

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

Sim> 1

--- 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}

which means for the given input *a* to process E0 the only possible output is *r*. E0 is the potentially active process and the others E1, E2 and E3 and presently idle. If the user selects this input by typing choice 1 the next available steps in the transitions are shown. Here there is a choice of alternative steps numbered 1 and 2. The notation t<r> indicates that the transition is an internal τ- transition [BUTL95b].

The following example output shows the power of the tool in running a simulation which finally gives a deadlock situation and its ablity then to reproduce the history of the conditions leading to the deadlock:

Simulated agent: (E0 | 'v.E1 | E2 | u.'b.E3)\{r,s,u,v}

Transitions:

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

Sim> 1

--- a --->

Simulated agent: ('r.E0 | 'v.E1 | E2 | u.'b.E3)\{r,s,u,v}

Transitions:

** Deadlocked. **

Sim> history

0: Eq43 --- a --->

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

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

3: ('r.E0 | 'v.E1 | E2 | E3)\{r,s,u,v} --- t<v> --->

4: ('r.E0 | E1 | E2 | u.'b.E3)\{r,s,u,v} --- t<r> --->

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

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

Other analyses can be performed such as exploring the size of a process and testing its equivalence to another process. For equality of processes the output can be obtained in modal calculus form:

Command: dfobs Buff3

Agent: Eq41

Identifier: Id2

false

Command: ppi Id2

proposition Id2 = <<a>> [[a]] [[a]] [[a]] F

This requires interpretation by the experienced user.

1.3 The Π-DFD Project

Data flow diagrams have been in use for many years as a tool in software development and a need was identified to be able to re-engineer them when legacy systems would be required to be updated. The structured analysis approach including the use of DFDs as a part of it has been found to be very useful because it is modular, graphic, top-down and implementation independent. It is for this reason it was considered that DFDs were worthy of further study and should be included in methods for re-engineering legacy systems The Π-DFD project was initiated at Concordia University [BUTL95a] and [BUTL95b] to

tackle this problem CWB would be a useful tool for analysis of these DFDs but as could been seen in the previous section, the user interface and CCS notation is not very user friendly. A tuple notation was devised to describe the DFD processes and dataflows and their relationships and there was the possibility that some of the information could be obtained by electronic scanning of paper documents.

An example of the tuple format is:

```
0: (SOURCE_TERMINATOR  T1 (( P1,a)))
   (SOURCE_TERMINATOR T2  ((P1,b)))
   (PROCESS P1 0  ((T3,c), (T4,D)))
    (SINK_TERMINATOR T3)
    (SINK_TERMINATOR T4)
```

(Also see Appendix A)

This describes a top level process with four external processes T1, T2, T3 and T4, one internal P1 with inputs a and b to P1 from the sources T1 and T2 respectively and outputs c and d from P1 to the sinks T3 and T4. Further levels of process refinement could be defined During the course of the project a translator was written which would transform the tuple format into CWB notation. Before the graphic interface was implemented, the tuple representation lacked positioning information required to represent the diagram graphically but it was felt that this could be supplied by a scanner at some future date.

1.4 The Π-DFD Graphic Interface

1.4.1 The need for an improved interface.

The purpose of this Master of Computer Science project was to provide a graphic interface to the Π-DFD system so that diagrams in tuple format could be displayed in graphic form and simulations could be performed via CWB and displayed dynamically The existing interface was not user friendly and had room for improvement

The CWB process was currently installed on the UNIX system and communication routines had already been set up between the translator and CWB It was decided therefore to implement a graphic interface on Sun workstations under UNIX and X-windows using the ET++ Development Environment for C++ [WEIN89]. The ET system provided a standard window for the interface and some built-in features such as easy specification of menu options and capture of mouse input Applications developed under ET would have built-in flow of control and event behavior: any pre-defined behavior could however be redefined through specialization by the user. The application developed would be in the Model-View-Controller model of object-oriented design.

1.4.2 The breakdown of roles

It was decided that as this was a joint project with Alison Greig there would be a broad split in the interface roles into two aspects:

1. Interpretation of the tuple representation into the display of the diagram graphics, an improved means of input of instructions to CWB and more easily comprehensible output with dynamic display of the simulation process

2 The communication between the interface and CWB and the internal maintenance of the status of the processes and actions

1.4.3 My Individual Role

More precisely my role in the project was as follows:

1. The tuple representation was to be changed to allow for positioning information from the layout of the diagram. The notation would change from, for example.

   (SOURCE_TERMINATOR T1 ((P1,a)))

   to

   (SOURCE_TERMINATOR T1 ((P1,a)) (100,120))

   where (100,120) was the coordinate position of the source process T1.

   I made the appropriate amendments to the existing translator to allow for this.

2. I also changed the translator to produce an intermediate file which could be more easily interpreted into the graphic diagram representation. This meant that the translator could be run once to produce a file in suitable format for the interface to interpret as output and simultaneously produce a file in CWB format which represented the diagram. (See Appendix A).

3. The existing translator only dealt with one particular tuple file so it had to be generalized to allow input from any tuple file.

4 For the graphic interface, I interpreted the intermediate file from the translator into the diagram graphic format, providing different graphic representation for terminator and internal processes and for the actions between processes I provided a variety of graphic highlight modes for different statuses of the processes and for the action lines as they would change during running of the simulation process.

5. Because this was a windows application, I had to decide what information was required to be stored in order to be able to redraw the current status of the diagram so that it was consistent and up-to-date

6. As the positions of the action lines are not specified in the tuple format these had to calculated based on the process positions Appropriate arrow positions for the leading endpoint also had to be calculated.

7. I specified how the simulation output was to be interpreted (Appendix B) and which functions would be required by the graphic part of the interface to be able to redraw the diagram's current status after a simulation step.

8. I intercepted and interpreted the mouse input for pursuing the simulation to its next step, communicating this to the ET system using its command system Both user control of the path step by step through the simulation as well as random selection of the path by the system was implemented.

9 The specification in section 2 was produced jointly. I wrote the initial specification and Alison Greig enlarged it with a lot of the details on the CWB commands I made further amendments to unify the style and to include the graphic notation for displayed diagrams. For section 3, I drew the global event diagrams, the architectural diagrams and the class diagrams for the user interface. I also wrote the detailed class definitions for the user interface classes Sections 1 and 4 and the appendices, glossary etc. were produced by myself

2 Specification for the 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 [MOLL 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

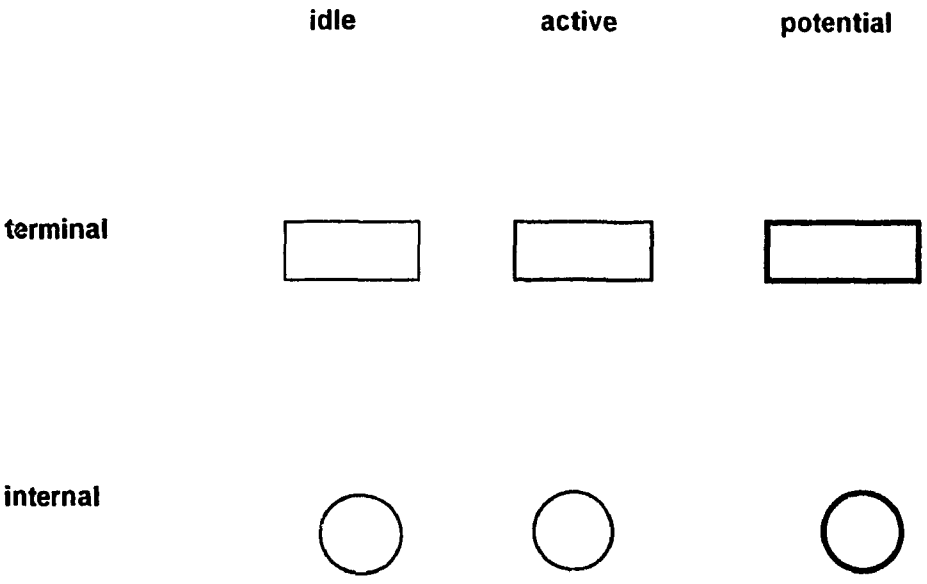
Conventions for graphic display of actions:

- 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

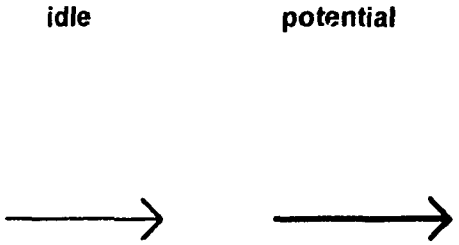
2.9.3.1 Processes

Conventions for processes:



2.9.3.2 Actions

Conventions for action lines:



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 a 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,
- 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 [BUTL95A, 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 process 1_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;
- (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.

2.10.6 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 The Π-DFD Graphic Interface with an idle diagram

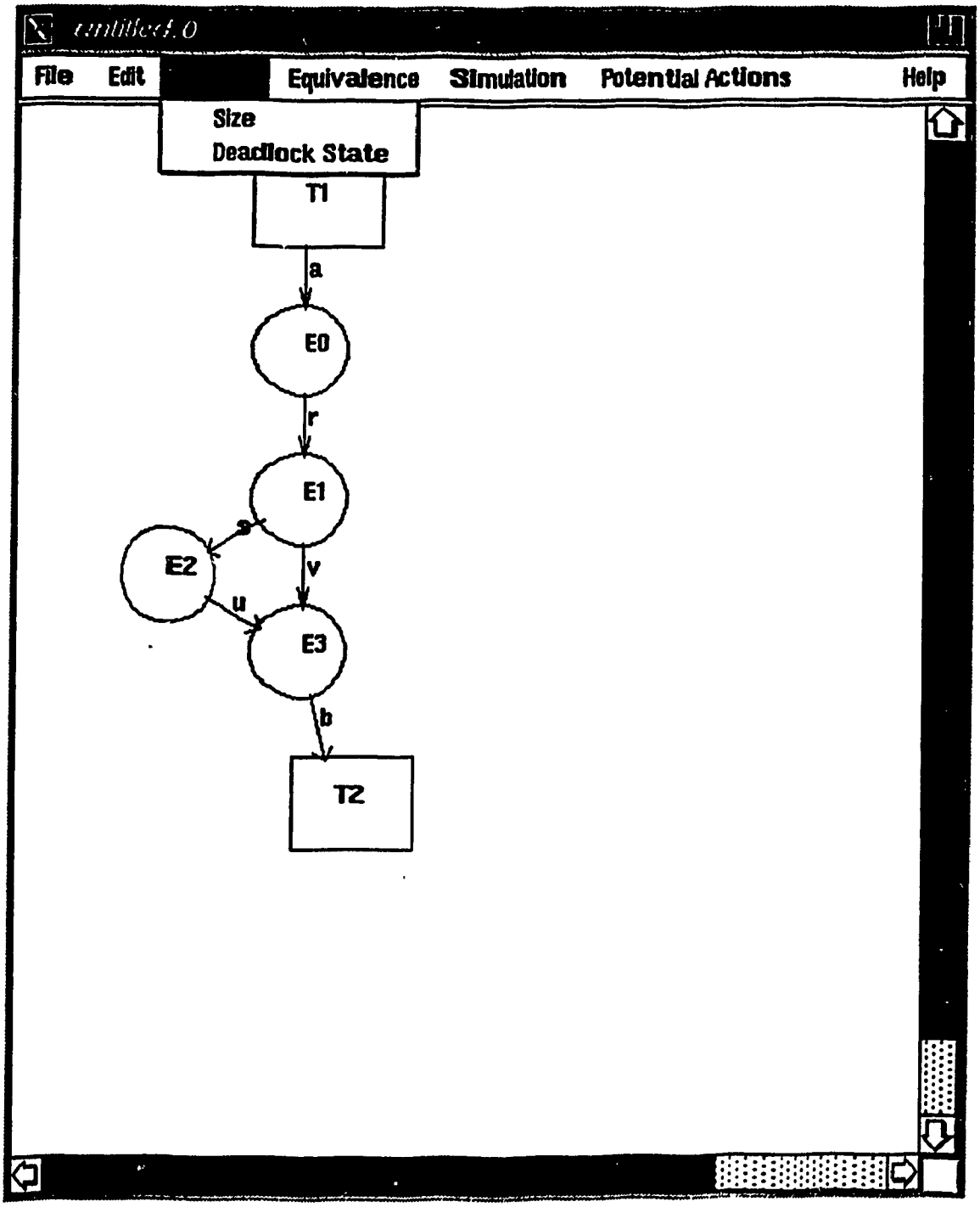
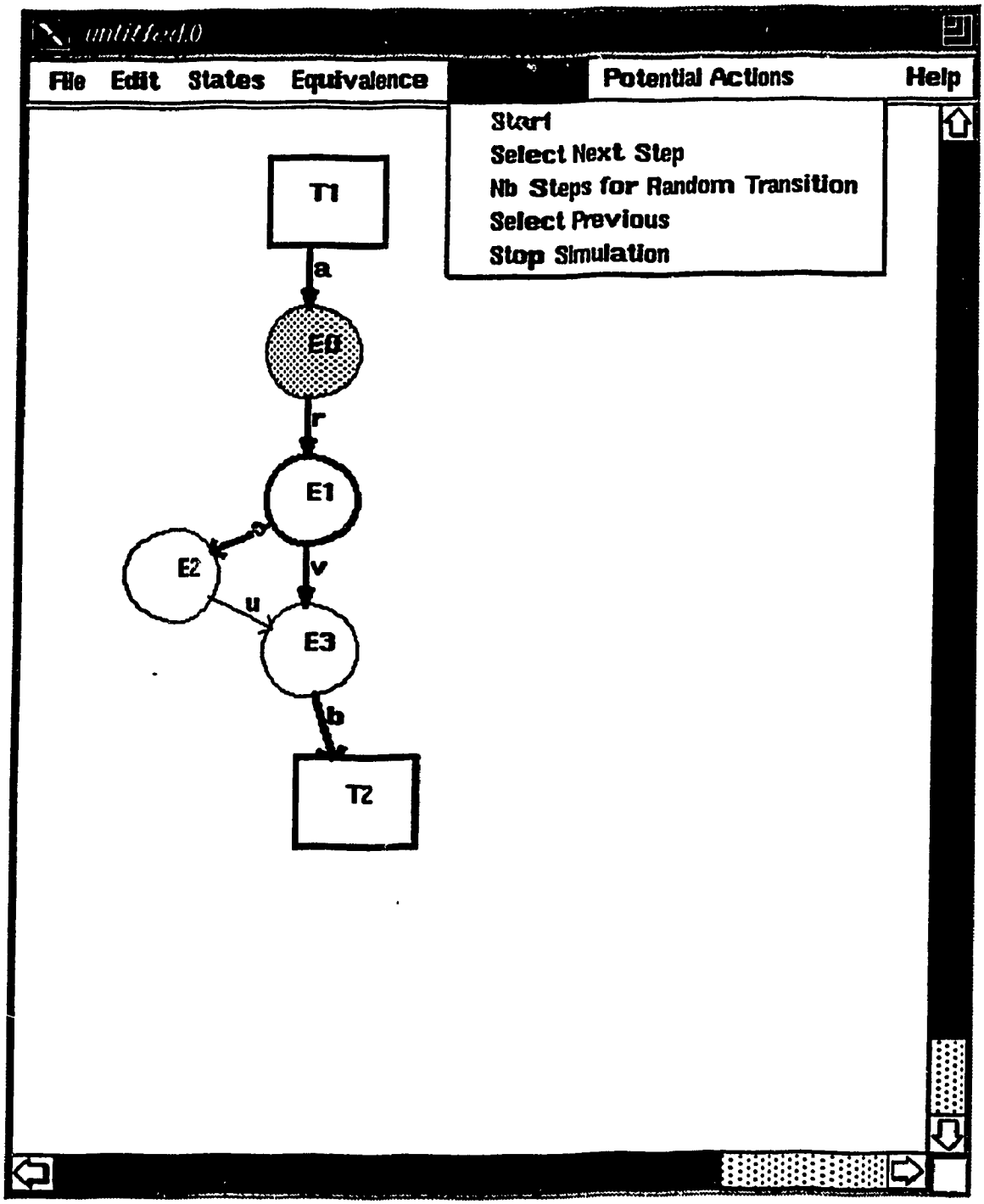


Figure 2.11.2 The Π-DFD Graphic Interface during a simulation



3. Design

3.1 Overview

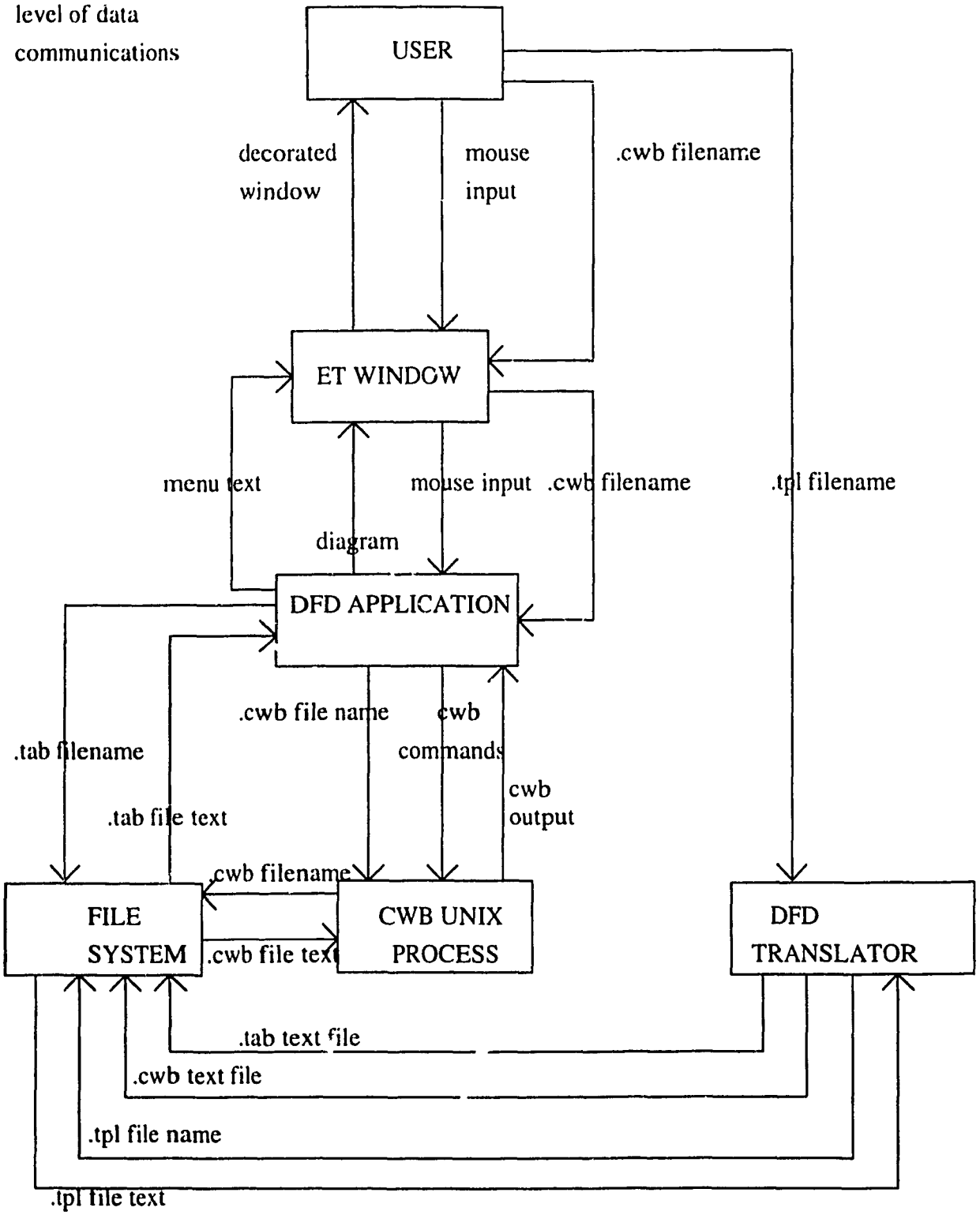
The DFD Application has been divided into two major components: the User Interface and the CWB Interface. Each of these are described at a high level in sections 3.1.3 [Class Diagrams:User Interface] and in section 3.1.4 [Class Diagrams:Cwb Interface]. All the classes are described in detail in sections 3.2 [Design: User Interface] and 3.3 [Design: Cwb Interface].

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.

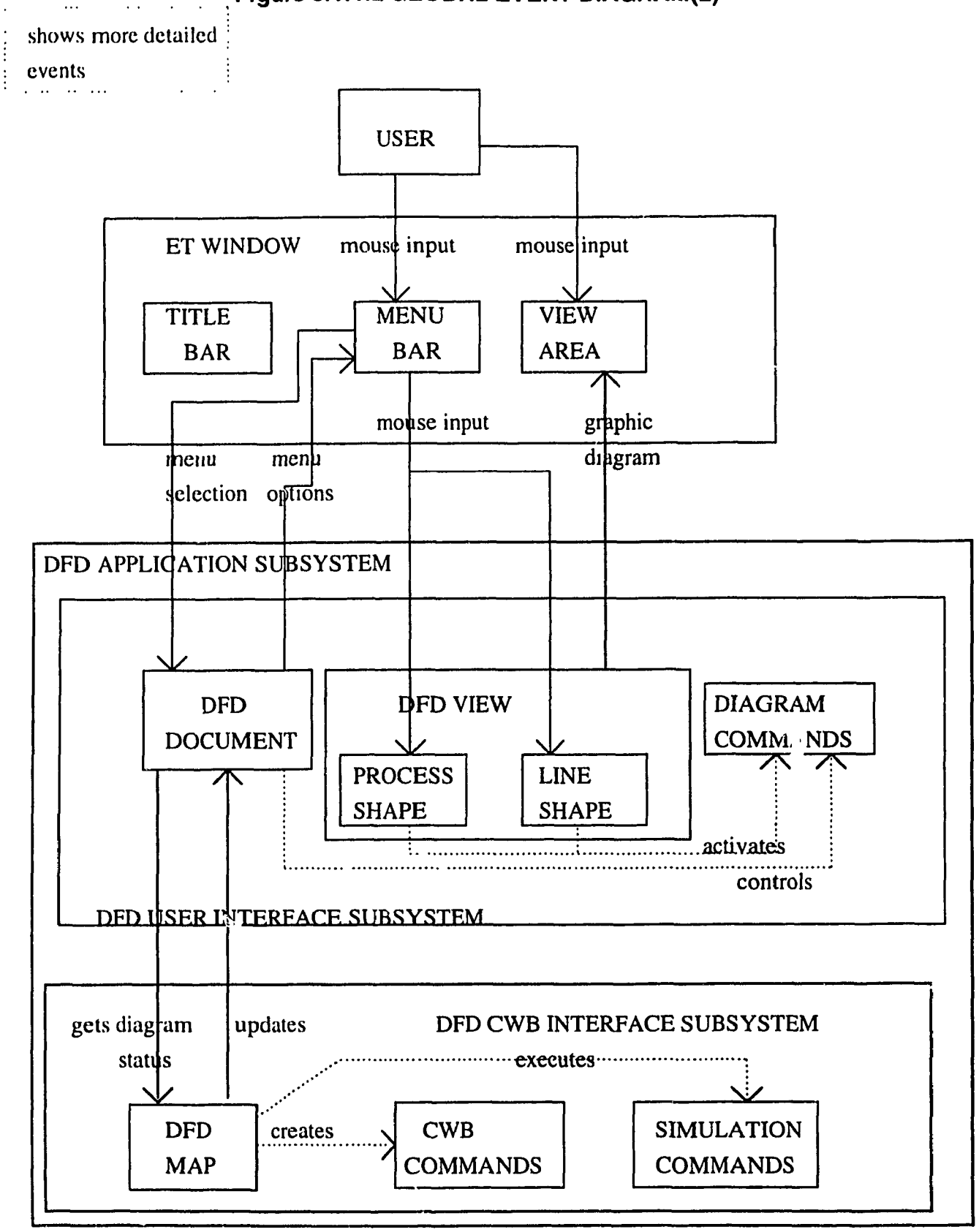
Figure 3.1.1.1 GLOBAL EVENT DIAGRAM (1)

shows the top level of data communications



The following diagram 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)

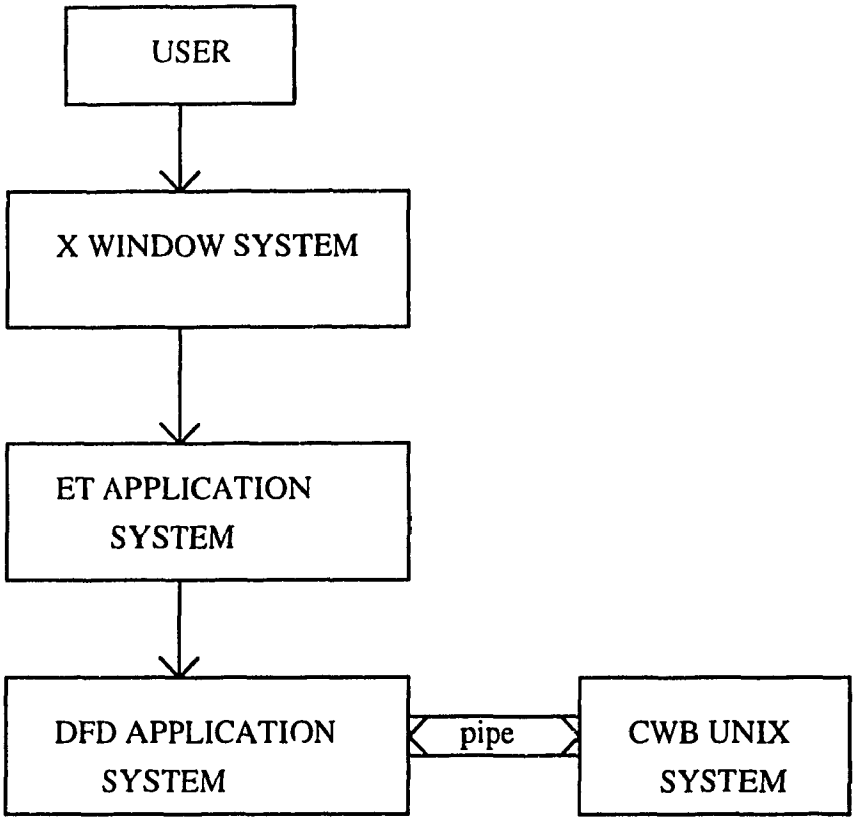


3.1.2 General Architecture

The following diagram shows the system topology i.e. the interdependence of the main subsystems and their hierarchical relationship:

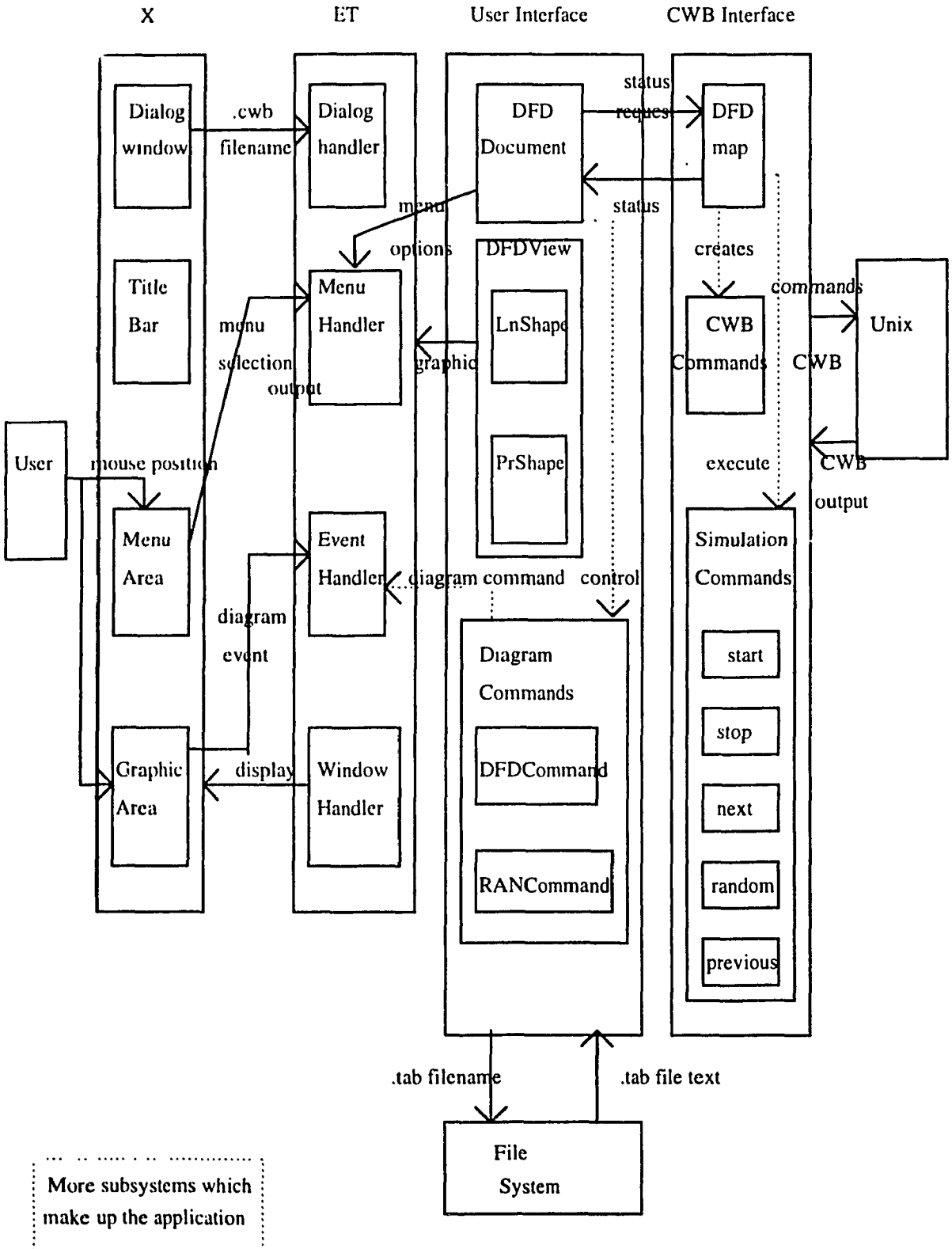
Figure 3.1.2.1 TOP LEVEL ARCHITECTURE

shows the topology
of the system



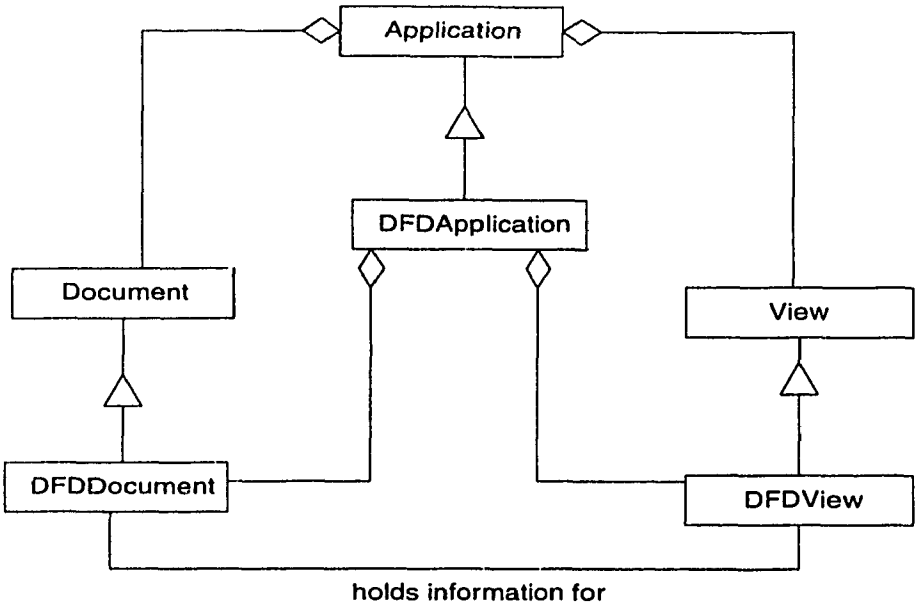
The following diagram shows a further breakdown of the major subsystems and the flow of data between them:

Figure 3.1.2.2 ARCHITECTURE -- SUBSYSTEMS



3.1.3 Class diagrams: User Interface

An overall class diagram is shown in the following diagram. It gives an overview of the relationship between the document, the view and the application.

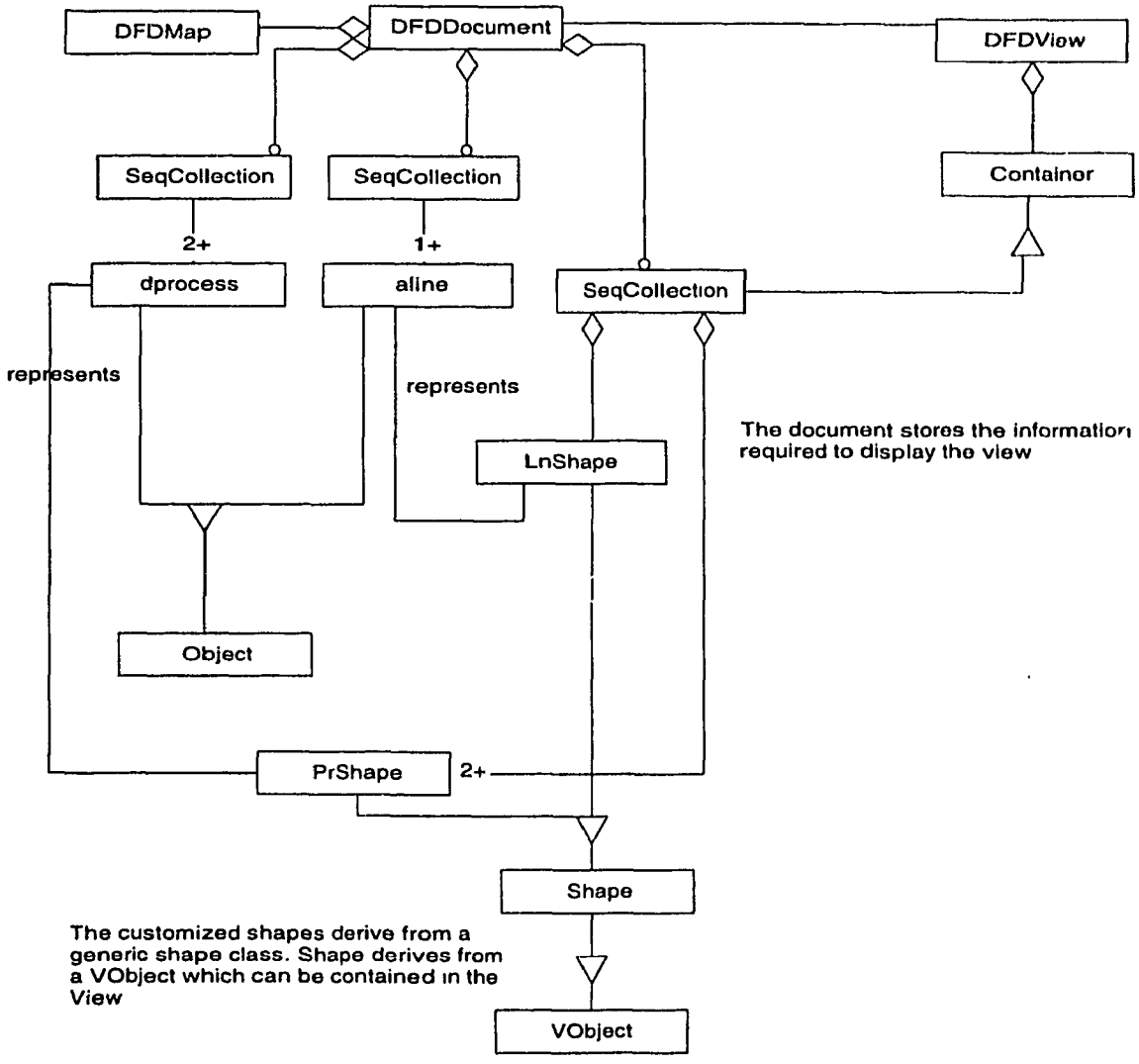


The customized application inherits standard default behaviour from the ET application framework

Figure 3.1.3.1 Overview class diagram

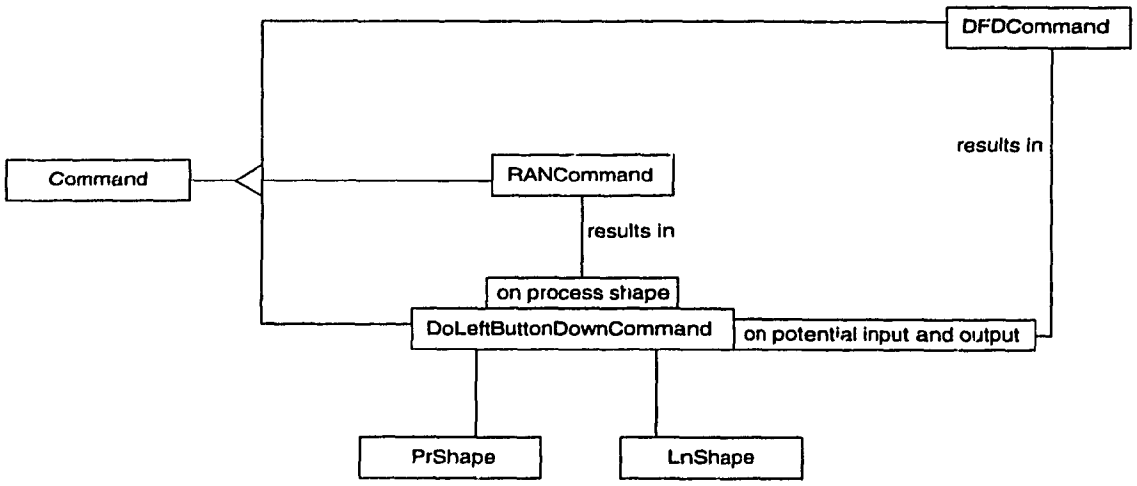
The following class diagram shows how objects representing the processes and actions (dprocess and aline) have Vobjects (which can appear in the view) associated to them. The Lnshape and Prshape classes are based on a generic Shape object so are collected into one sequence collection. This sequence collection is associated with the View and the information contained in it is used to redraw the DFDView window.

Figure 3.1.3.2 Classes for generation and display of diagram shapes



The diagram command classes RANCommand and DFDCommand 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



When the user mouse clicks on the diagram, the program responds differently depending on where the interaction occurs.

In simulation mode, one click on a potential process gives random simulation: a click on a potential input followed by one on a potential output gives the next simulation step.

3.1.4 Class Diagrams: 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 and does the 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 dataflow diagram. PotentialAction represents the actions that can be made on the dataflow diagram while in simulation mode. This information is displayed on the dataflow diagram 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

This class is simple and provides input and output to the CWB process. It is held by DFDMap (see section 3.1.4.2.1 DFDMap Class Aggregation). It's functionality is described in detail in section 3.3.1 Cwb Process.

3.1.4.2 Status of the Dfd

The status of the dataflow diagram is represented with these objects. The *current* or most recent status includes the status of the processes and actions in the dataflow diagram.

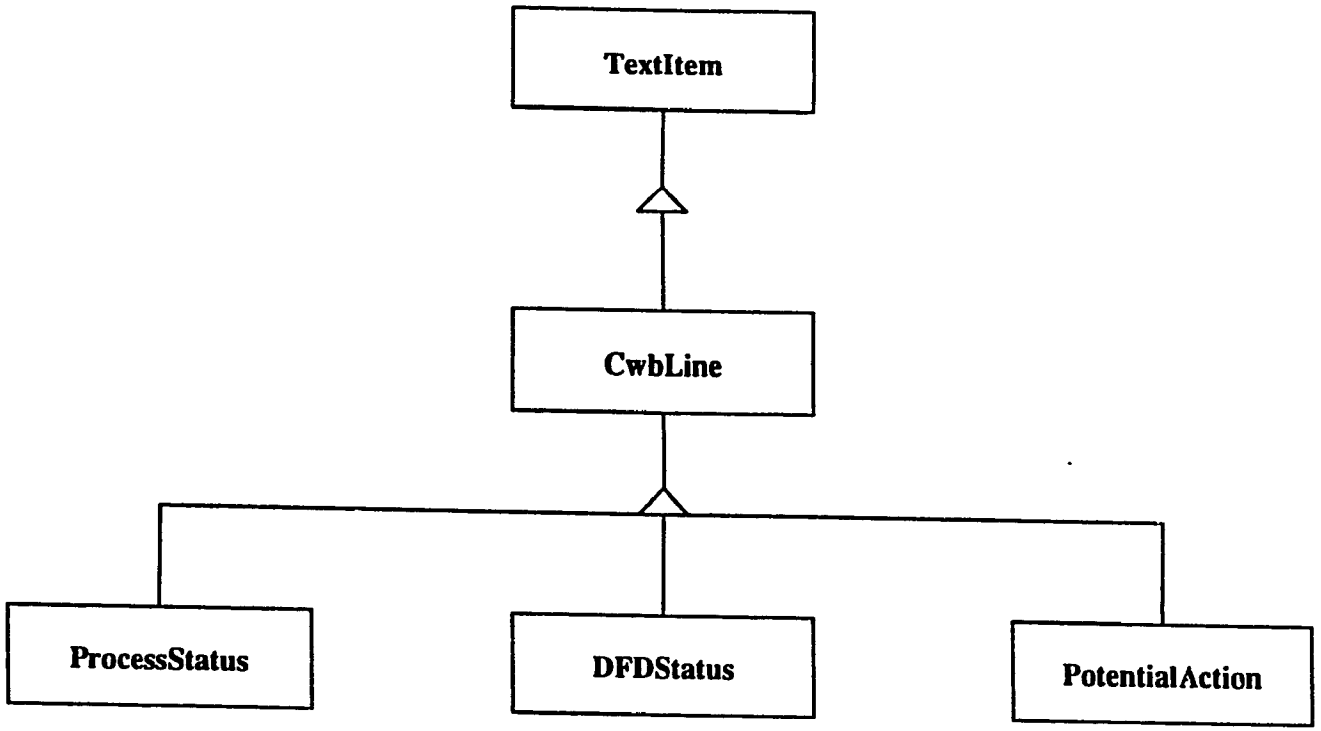


Figure 3.1.4.2.1: DataFlow Diagram Status Classes Generalization

This is ProcessStatus. The status of all the elements in the dataflow diagram is stored in DFDStatus. When a dataflow diagram is in simulation mode, it has potential actions, represented with PotentialActions. These statuses are held by the DFDMap. The relationship between the 3 status classes can be seen in figure 3.1.4.3.1: DFDMap Class Aggregation.

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: DataFlow Diagram Status Classes Generalization.

3.1.4.3 Interface between Application and CWB

The DFDMap is the class that separates the CwbInterface from the rest of the application [figure 3.1.4.3.1: DFD Map Class Aggregation]. It starts the CwbProcess, it creates all CwbCommands and SimCommands, it updates and holds the status information for the DFD, and it tells the DFDDiagram that a change in status has occurred. (This can be seen in figure 3.1.4.5.3: SimCommand Events).

The DFD Map holds the status information for the DFD diagram. It contains one current DFDStatus which describes the status of each process in the DFD. It is generated each time there is a change made through a SimCommand. 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. These are also generated each time there is a change made through a SimCommand.

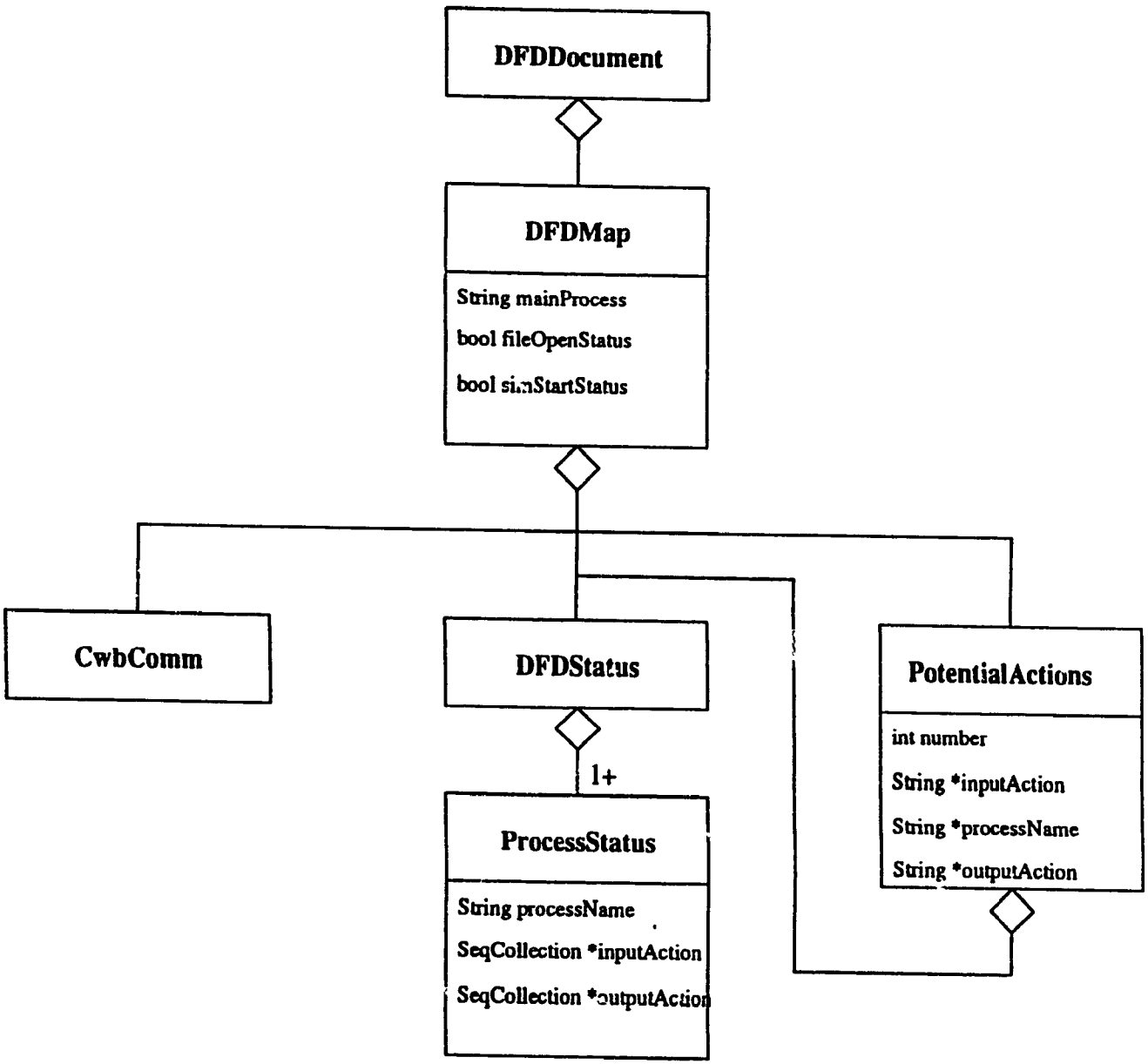


Figure 3.1.4.3.1: DFD Map Class Aggregation

3.1.4.4 Commands that interact 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. The menu items Select Next Step and Nb Steps for Random Transition can also be done by selecting an item directly on the DFD diagram.

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. The diagram [CwbCommand Classes Generalization] show 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 leaves 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.

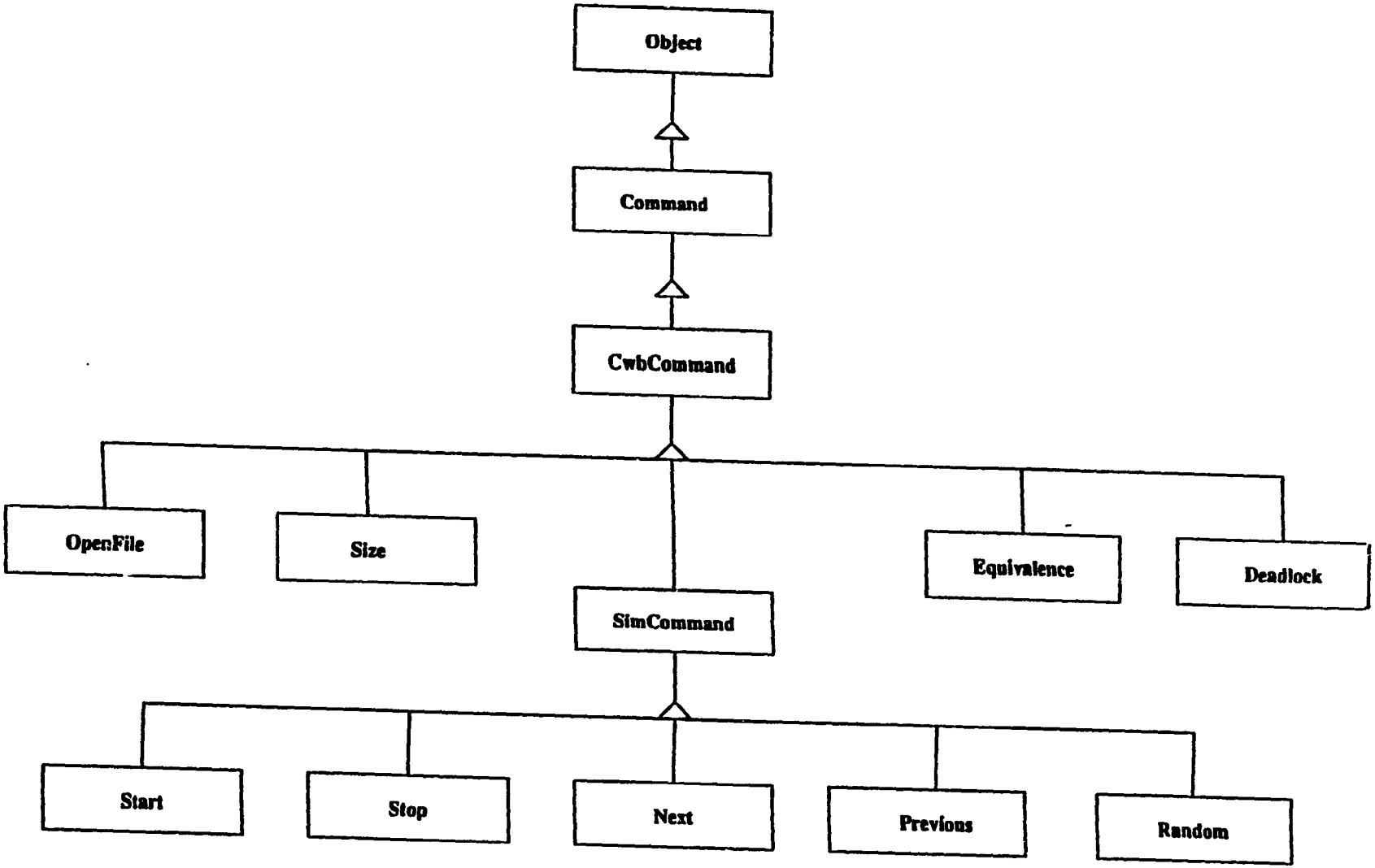


Figure 3.1.4.4.1: Cwb Command Classes Generalization

The interpretation for SimCommands is more complex and the Cwb process output is used to generate the DFDStatus and PotentialActions of the DFD.

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. The figure 3.1.4.5.1: DFD Events: User Input shows how these two types of commands are created.

Figure 3.1.4.5.2: CwbCommand Events 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 [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 SimCommand Events 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.

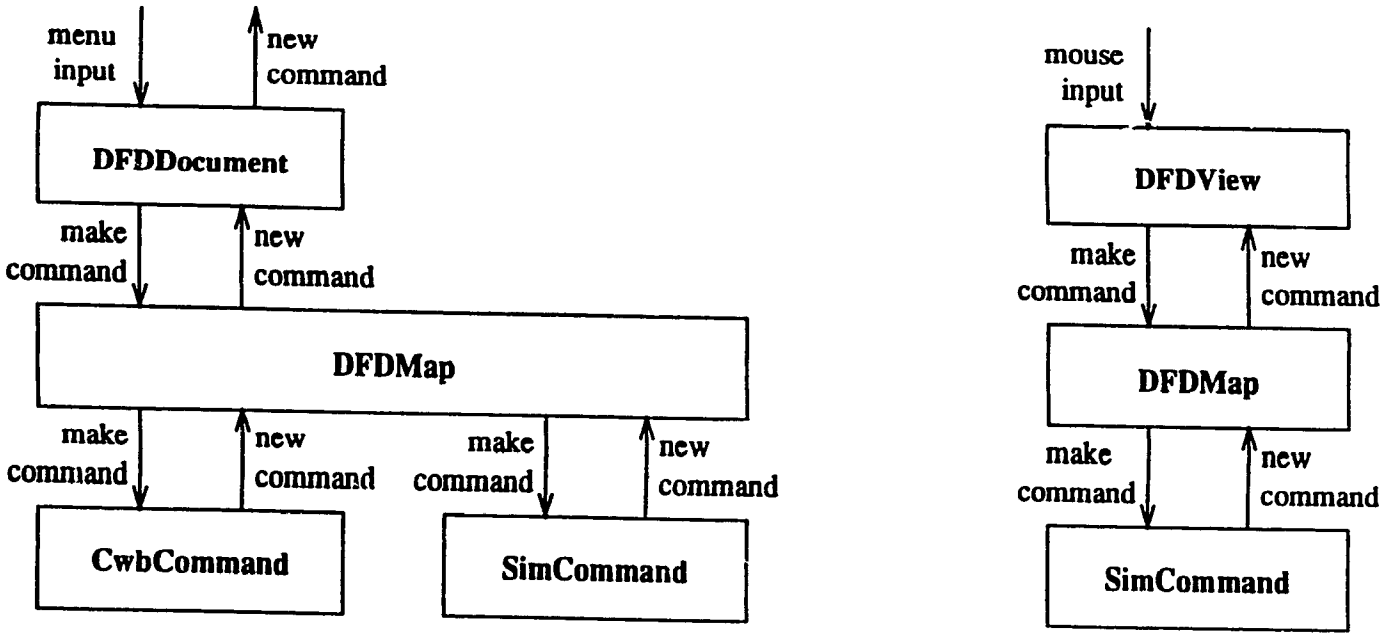


Figure 3.1.4.5.1: DFD Events: User Input

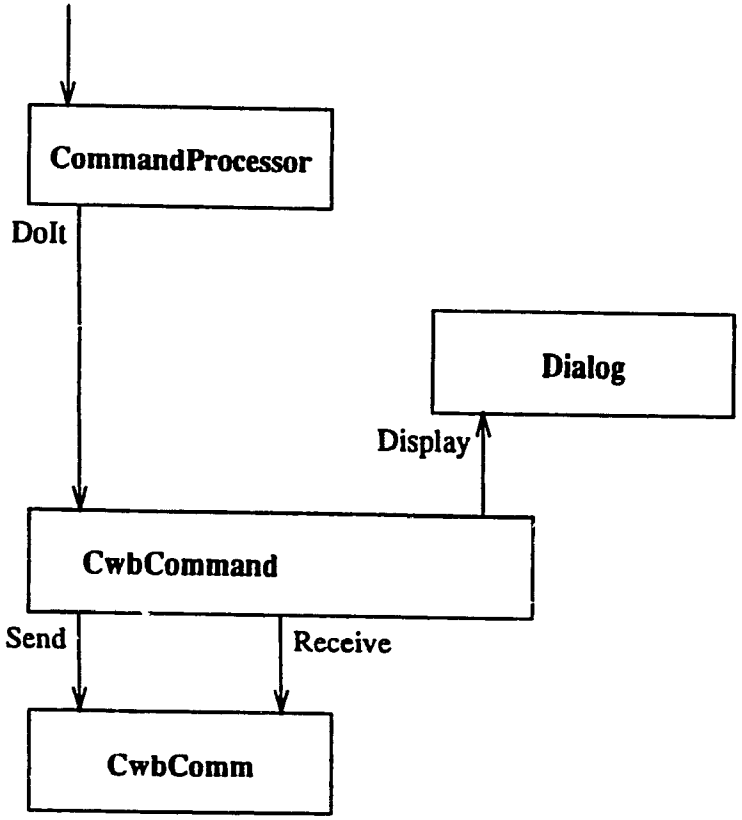


Figure 3.1.4.5.2: CwbCommand Events

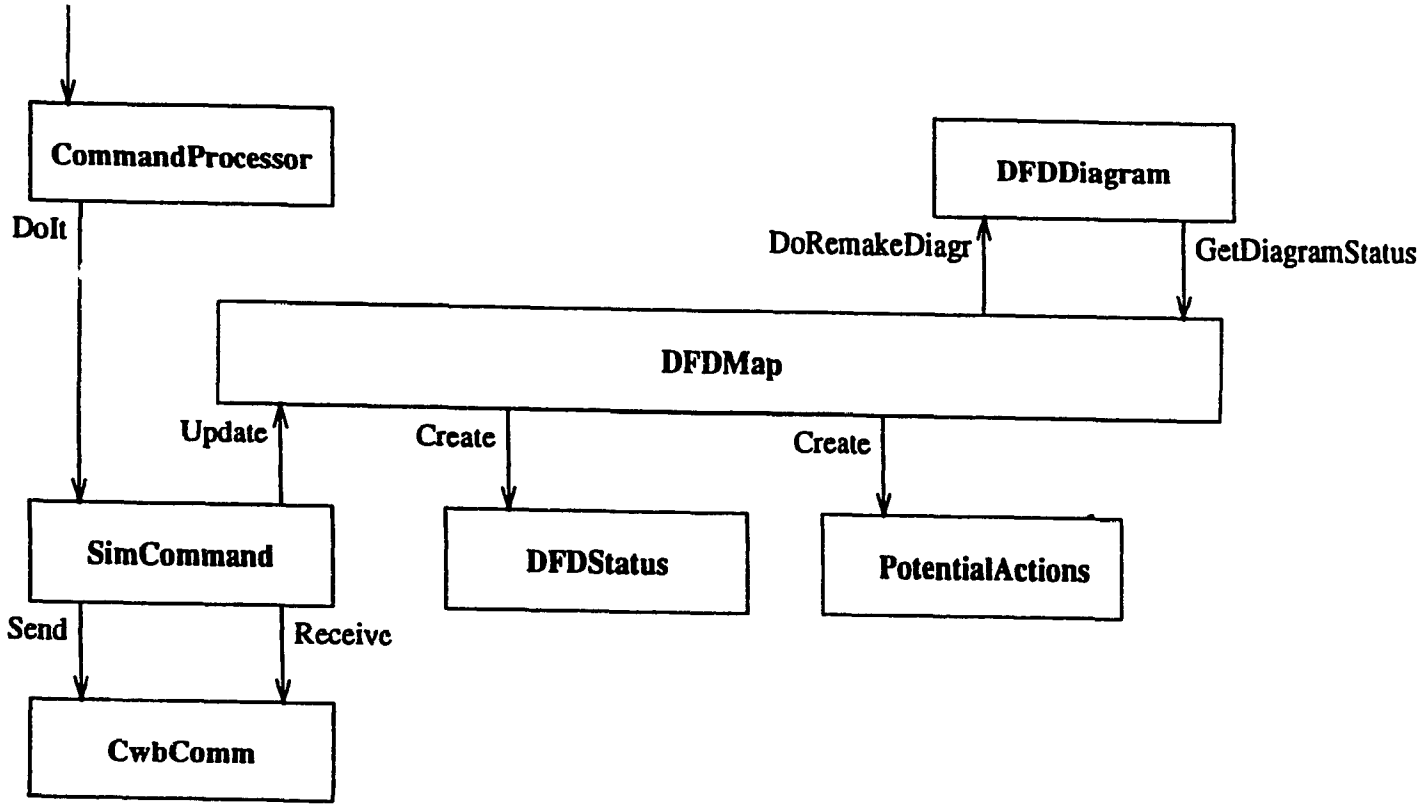


Figure 3.1.4.5.3: SimCommand Events

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 lab1	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.

OStream *&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 location p is contained in the LnShape object then the event is passed to method *DispatchEvents* in the *DFDView* and a user defined command (here DFDCommand) 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*, s: 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 location Point p is contained in the PrShape object then the event is passed to method *DispatchEvents* in *DFDView* and a user defined command (here RANCommand) 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 fro individual commands and know about the syntax of the commands for both the input and output for 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 2 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[MOLL92]

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 that want to take

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. Conclusion

4.1 Success of the graphic interface

4.1.1 Success in terms of general interface design

4.1.1.1 Compatibility

The functionality provided in the system with the graphic interface replaces directly many of the existing ones in the textual interface of CWB. Not all of the functionality has been incorporated on this project but it has the further capability to be easily extended to do so

4.1.1.2 Familiarity

The user who is familiar with current windows operating systems on PCs or Unix will immediately feel at home with the menu input and mouse as a control medium.

4.1.1.3 Simplicity

Mouse and menu-driven capability is a well-recognized means of simple system access

4.1.1.4 Direct Manipulation and control

Direct manipulation via the mouse and menus and control of the simulation via on-screen input gives users a sense of control which will make them feel comfortable with the interface.

4.1.1.5 Visual Presentation

The WYSIWIG (What You See Is What You Get) approach to the diagram presentation and the visual representation of the simulation as it happens should give the user a better feel for the process being analyzed.

4.1.1.6 Flexibility and ease of use

The interface makes the access to CWB available to a wider range of users because there is less notation to be learned before using it.

4.1.1.7 Consistency

A good user interface should be consistent with other common applications. The use of the ET Application framework gives the interface a consistency with other ET applications developed with ET the pulldown menus of the ET system, window presentation and use of the mouse are common communication tools for users of applications in PC Windows and X-windows systems and so are familiar to many users.

4.1.2 Success as an improved means of communicating with CWB

I felt that the graphic interface was a successful additional component for the Π-DFD project. The user is shielded from those parts of the system which are not user friendly: the tuple representation conforms more to the visual model of the diagram than does the CCS, while the translator performs the work of conversion to the CCS notation for CWB Access is given to the powerful analytical capabilities of CWB without requiring that the user learn the CCS notation or commands. The translation process need only be performed once to create the intermediate files for any one diagram.

The menu capability is a far more intuitive means of communicating the CCS commands for diagram loading, comparison, analysis and simulation than the textual means. The DFD itself is displayed similar to its original paper format and the simulation output is visually and dynamically displayed so that the user can observe the activity as it happens. Being able to control the simulation either step by step or by random selection of next step

choice and to have the result presented immediately in diagrammatic form gives the user a far better sense of the process capabilities than when textual output must be interpreted

4.2 Evaluation of the development environment

4.2.1 Advantages

Many built-in facilities in ET simplify the development of a useful tool with a user interface consistent with other applications. The display of the window, provision of scrollbars and menus and automatic event loop control are all provided under ET and save the user a lot of work in application development. There is much default behaviour of the in-built operations and events that the user is spared from providing The Model View Controller (MVC) model of program design gives a good framework for a strong design in the object-oriented application. Application frameworks ..

> *promote an application architecture that embodies the software engineering principles of loose coupling and strong cohesion*

[OBRI95, p. 84]

and this structure (loose coupling/ strong cohesion) should

> *allow us to see modules as black boxes when the overall structure of the system is described*

[GHEZ91, p. 51]

The SNIFF development environment provides editors, project and object management tools. In fact I found the application project setup far from intuitive and did not use this in

the development but the ability in the editor to branch between classes and the object browsing capabilities made finding out about the ET class mechanisms much easier.

4.2.2 Disadvantages

Developing an application successfully under the ET Application framework seemed to require a lot of knowledge of the hierarchy of the classes and the event and command handling. For example using the class hierarchy, the default behaviour for the DoLeftButtonDownCommand on the View could be overridden by redefinition on any of the view objects. In this way behaviour for selecting different view objects could be customized. This was a very powerful facility provided by ET but one which required detailed knowledge of the internal organization to make best use of it. The same was true for the understanding of the command and event dispatching control and for the precise bounds for selecting view objects with the mouse.

4.3 Future improvements and enhancements

4.3.1 Multiple levels of diagram decomposition

It would be a useful enhancement for the comparison of diagrams and equivalencing their decompositions for re-engineering purposes to be able to display multiple levels of the diagram. There is already a notation for this in the tuple representation: a tree structure would be suitable to provide the mechanism for moving around in the diagram level hierarchy.

4.3.2 Multiple Windows

The ability to open multiple child windows from one parent application could be used to compare and contrast two diagrams , each in a separate window, or to display different levels of the same diagram in different windows.

4.3.3 More precise mouse selection of objects

The default method in ET for defining the region within which mouse selection is valid is to define a rectangle around an object. In a large diagram with lines and processes in close proximity, this could be refined more precisely to a more limited area. Particularly for the action lines, depending on the angle, the bounding rectangle which is the default, might be refined to a boundary closer to the line.

4.3.4 More CWB facilities

Some CWB facilities were not included such as derivation commands to display observable actions, reachable agents etc. Further menu options could provide these.

4.3.5 Diagram editing

ET was designed for easy development of graphic editing facilities so this could be capitalized on to make a graphic editor for dataflow diagrams.

4.3.6 File mechanisms

The file dialogue mechanism we implemented was via a general popup window. With further knowledge now of the ET mechanisms there is a built in file dialogue input facility

which would have been more consistent This was a problem caused by lack of documentation for ET

4.4 What was learned from the project

The project was a very useful experience, exposing me to a new application framework. It demonstrated to me the power of a framework to provide a lot of standard functionality and management of the interaction between the subsystems of the application.

We explored the CWB analytical engine and discovered its powerful mechanisms for diagram analysis and simulation. The translator had to be understood before I could alter the code to incorporate the process coordinate positions: it showed me an example of the tokenization of input as a means of parsing and how one Unix process could communicate via pipes to another Unix process.

For the graphics methods for the visual object shapes I had to decipher the ET graphics routines and learned the mechanism for mouse selection of objects and how the area of selection close to an object was defined. For selection of the shapes on-screen I created user defined commands to define the behaviour when shapes were selected during simulation.

The lack of documentation for ET and uncommented code brought home very strongly to me the need for good user manuals and documentation. Initially we examined the Borland graphic library with a view to possibly using this for the project: I had used this previously

learning only from the manual which explained very clearly the graphics functions. the only documentation I had for the corresponding ET elements was the program code which in addition was entirely uncommented.

Despite this these potential obstacles, it was very satisfying to achieve what I felt was a considerable improvement of the existing interface and to work on a dynamic windows interface.

References

[BENN95] Bennett K Legacy Systems. coping with success. *IEEE Software*. Los Alamitos, CA. IEEE Computer Society, January 1995, pp. 19-23.

[BUTL95a] Butler G,Grogono P, Shinghal R & Tjandra I. Analyzing the logical structure of data flow diagrams in software documents. *Proceedings of the Third International Conference on Document Analysis and Understanding*, Montreal. IEEE Press, August 14--16, 1995, pp. 575--578.

[BUTL95b] Butler G,Grogono P, Shinghal & Tjandra I. Retrieving information from data flow diagrams. In Wills L, Newcomb P & Chikofsky (eds*). Proceedings of Second Working Conference on Reverse Engineering*,(Toronto, July 14--16, 1995). Los Alamitos, CA: IEEE Computer Society Press, 1995, pp. 22--29.

[BUTL95c] Butler G, Grogono P, Shinghal R & Tjandra I. *Knowledge and the Recognition and Understanding of Software Documents*. Department of Computer Science, Concordia University, Montreal, Quebec, Canada. February 1995, pp. 1-47.

[CHEN92] Chen M-J & Chung C-G. Preventative Structural Analysis of Dataflow Diagrams. *Information and Software Technology*, Vol. 34 No 2. London,UK: Butterworths,February 1992, pp. 117-130.

[DEMA79] DeMarco T . *Structured Analysis and System Specification*, New York: Yourdon, 1979.

[GHEZ91] Ghezzi C, Jazayeri M & Mandrioli D. *Software Engineeering*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[ILOG91] i-Logix Inc. *The Languages of Statemate*. Burlington, MA: January 1991.

[ILOG92] i-Logix Inc. *Statemate 4.5 User Reference Manual*. Burlington,MA: August 1992.

[MOLL92] Moller F. *The Edinburgh Concurrency Workbench (Version 6.1).* Department of Computer Science, University of Edinburgh, October 1992.

[OBRI95] O'Brien L. C++ Application Frameworks . *Software Development.* October 1995, pp. 84-89.

[RUMB91] Rumbaugh J, Blaha M, Premerlani W, Eddy F & Lorensen W. *Object-oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall, 1991.

[WEIN89] Weinand A,Gamma E & Marty R. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 2 , 1989, pp. 63-87.

[WOOD90] Woodman M Yourdon Dataflow Diagrams. In Ince D. & Andrews D. , *The Software Life Cycle.* Boston: Butterworth, 1990, pp. 129-167.

[YOCO79] Yourdon E. & Constantine L. *Structured Design: Fundamentals of a Discipline of Computer program and Systems Design* Vol.1. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[YOUR79] Yourdon E. *Managing the Structured Techniques.* Englewood Cliffs, NJ: Prentice-Hall, 1979.

Appendix A

Example data files

An example .tpl file:

```
0: ((SOURCE_TERMINATOR T1 ((E0,a)) 150 50)
  (PROCESS E0 1 ((E1,r)) 150 180)
  (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 150 370))
```

The translator produces an intermediate .tab file:

```
T1 1 0 150 50 E0 a
E0 0 0 150 180 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 0 150 370
```

and a .cwb file:

```
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
```

Appendix B

Requirements for translation of CWB Simulation output

This appendix describes the specification requirements for the interpretation of the CWB output in simulation mode. An example of which is given here:

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

GetPotActIn(OrdCollection * a)

Finds the potential inputs from the simulation output from CWB.

Algorithm:

The potential inputs are given by the action label contained between the dashes e..g --- a --- shows a to be the potential input. For all numbered output lines save the potential inputs in the Collection a. If the character string is of the form t<a> for a τ-transition, strip off the t<> characters before adding to the Collection.

GetPotActOut(OrdCollection *b)

Finds potential input actions

Algorithm:

For each numbered CWB simulation output line, find the actions x which appear as 'x.P to the right of the arrow where x is not the action contained in the --- (action) --- > string. Add the found action name x to b.

GetPotActAgent(OrdCollection * c)

Finds potential processes from the CWB simulation output.

Algorithm:

For each line of simulation output, an active process P is one which appears as u.'x.P on the right hand of the arrow --- > or else as 'x.P where x is not the action label contained in -- - - >. Add the found process name P to c.

GetActiveAgent(OrdCollection *d)

Determine active states from map process statuses.

Glossary

agent	a process which handles data
CASE tool	Computer Aided Software Engineering tool
CWB	Concurrency Workbench
dataflow	the passage of data between processes
datastore	a permanent repository for data
DFD	Dataflow Diagram
ET	An application framework for developing C++ programs
SNIFF	A project development environment
statechart	a chart which shows dynamic behavior
(T)CCS	(Temporal) Calculus of Communicating Systems
translator	a program which translates from the .tpl file format to the .cwb and.tab format
X windows	a windowing system used on UNIX