University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

1998

# Semantics-driven dataflow diagram processing.

Lizhong Zhou
*University of Windsor*

Follow this and additional works at: https://scholar.uwindsor.ca/etd

## Recommended Citation

Zhou, Lizhong, "Semantics-driven dataflow diagram processing." (1998). *Electronic Theses and Dissertations*. 3425.
https://scholar.uwindsor.ca/etd/3425

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Semantics-Driven Dataflow Diagram Processing

by

**Lizhong Zhou**

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree
of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
1998

**© Lizhong Zhou**

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

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

0-612-52496-5

Canada

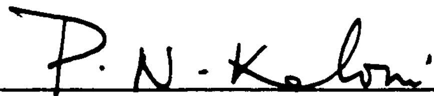
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**APPROVED BY:**

Dr. Yung H. Tsin, Chair, Computer Science

Dr. Indra A. Tjandra, Supervisor, Computer Science

Dr. Young Park, Department Reader, Computer Science



Dr. Purna N. Kaloni, External Reader, Department of Mathematics

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Abstract

Dataflow diagram is a commonly used tool of structured analysis and design techniques in specifications and design of a software system, and in analysis of an existing system as well. While automatic generating dataflow diagrams saves system designers from tedious drawing and help them develop a new system, simulating dataflow diagrams provides system analysts with a dynamic graph and help them understand an existing system. CASE tools for dataflow diagrams play an important role in software engineering. Methodologies applied to the tools are dominant issues extensively evaluated by tools designers. Executable specifications with dataflow diagrams turn out an opportunity to execute graphic dataflow diagrams for systems analysts to simulate the behavior of a system.

In this thesis, a syntax representation of dataflow diagram was developed, and a formal specification for dataflow diagram was established. A parser of this developed CASE tool translates the syntax representation of DFDs into their semantic representation. An interpreter of this tool then analyzes the DFDs semantic notations and builds a set of services of a system represented by the DFDs. This CASE tool can be used to simulate system behavior, check equivalence of two systems and detect deadlock. Based on its features, this tool can be used in every phase through entire software life cycle.

***Keywords:*** dataflow diagrams, software life cycle, software reuse, structured analysis and design techniques, software specification documents and design documents, formal specifications, grammar, CASE tools, CCS, Java.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

*To My Parents and My Wife*



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# Acknowledgements

I would like to express my sincere thanks and respect to my supervisor Dr. Indra Tjandra. His guidance and enthusiasm made this thesis an enjoyable experience for me. Many thanks to Dr. Young Park for his valuable advice and comments. I would also like to thank Dr. Purna Kaloni for his consistent support.

I would like to extend my thanks to Dr. Subir Bandyopadhyay for his wonderful suggestions and to Dr. Yung Tsin for chairing the defence committee.

My special thanks go to system administrator Walid Mnaymneh and secretary Mary for their help.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# TABLE OF CONTENTS

Abstract . . . . . . . . . . iii

Acknowledgements . . . . . . . . . . v

1 INTRODUCTION . . . . . . . . . . 1

1.1 The Role of DFD in Software Life Cycle . . . . . . . . . . 2

1.1.1 DFDs in Software Design . . . . . . . . . . 2

1.1.2 DFDs in Software Maintenance . . . . . . . . . . 3

1.1.3 DFDs in Software Reuse . . . . . . . . . . 5

1.1.4 DFDs in Reverse Engineering . . . . . . . . . . 7

1.2 The Role of DFD in the Object-Oriented Paradigm . . . . . . 9

1.3 Overview of the Approaches of DFD Processing . . . . . . . 10

1.4 The Problems with Previous DFD Tools . . . . . . . . . . 13

1.5 The Organization of the Thesis . . . . . . . . . . 15

2 DATA FLOW DIAGRAMS . . . . . . . . . . 16

2.1 Overview . . . . . . . . . . 16

2.2 The Components of DFDs . . . . . . . . . . 18

2.2.1 The Process . . . . . . . . . . 19

2.2.2 The flow . . . . . . . . . . 19

2.2.3 The Store . . . . . . . . . . 21

2.2.4 The Terminator . . . . . . . . . . 22



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3 Constructing DFDs . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.3.1 Leveling and Balancing . . . . . . . . . . . . . . . . . . . . 25
2.3.2 Repartitioning . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.3.3 Evaluating and Improving DFD . . . . . . . . . . . . . . 28
3 CASE TOOLS FOR DFDS . . . . . . . . . . . . . . . . . . . . . . . 30
3.1 The Role of CASE Tools in Software Development . . . . . 30
3.2 Important Features in CASE tools . . . . . . . . . . . . . . . . 32
3.2.1 Graphics Support . . . . . . . . . . . . . . . . . . . . . . . 32
3.2.2 Error-Checking Features . . . . . . . . . . . . . . . . . . . 32
3.2.3 Cross-Checking of Different Models . . . . . . . . . . . 33
3.2.4 Additional Software Engineering Support . . . . . . . 34
3.3 Current CASE Tools for DFDs . . . . . . . . . . . . . . . . . . 35
3.3.1 DFD Editor and Processor . . . . . . . . . . . . . . . . . 35
3.3.2 Automatic Generation of Dataflow Diagrams . . . . . 44
3.3.3 Executable Dataflow Diagrams . . . . . . . . . . . . . . 47
3.3.4 Executing Dataflow Diagrams . . . . . . . . . . . . . . . 50
3.4 Formal DFD Specifications in CASE Tools . . . . . . . . . . 52
4 SEMANTIC DESCRIPTION OF DFDS . . . . . . . . . . . . . . . . 55
4.1 Calculus of Communicating System . . . . . . . . . . . . . . . 55
4.1.1 Sequencing . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.1.2 Choice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
4.1.3 Parallel Composition . . . . . . . . . . . . . . . . . . . . . 57
4.2 Semantic Representation of DFDs . . . . . . . . . . . . . . . 58



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5 A SEMANTIC-DRIVEN DFD PROCESSING SYSTEM . . . . . . 61
6 SYSTEM SPECIFICATION . . . . . . . . . . . . . . . . . . . . . . . . 63
6.1 General Description . . . . . . . . . . . . . . . . . . . . . . . . 63
6.1.1 The Purpose of DFDPRO . . . . . . . . . . . . . . . . . 63
6.1.2 Product Functions . . . . . . . . . . . . . . . . . . . . . 64
6.2 System Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.2.1 The Logical Structure of the System . . . . . . . . . . 65
6.2.2 Display Description . . . . . . . . . . . . . . . . . . . . 66
6.2.3 The Workspace . . . . . . . . . . . . . . . . . . . . . . . 67
6.3 System Services . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.3.1 Functional Requirements . . . . . . . . . . . . . . . . . 68
6.3.2 Translation Services . . . . . . . . . . . . . . . . . . . . 69
6.3.3 Simulation Services . . . . . . . . . . . . . . . . . . . . 79
7 SYSTEM DESIGN . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
7.1 High-Level Description . . . . . . . . . . . . . . . . . . . . . . 86
7.2 Design Refinement . . . . . . . . . . . . . . . . . . . . . . . . . 88
7.2.1 Graphic User Interface . . . . . . . . . . . . . . . . . . 88
7.2.2 Syntax Processor . . . . . . . . . . . . . . . . . . . . . . 89
7.2.3 Semantic Processor . . . . . . . . . . . . . . . . . . . . 90
7.2.4 Service Builder . . . . . . . . . . . . . . . . . . . . . . . 90
7.2.5 Display Constructor . . . . . . . . . . . . . . . . . . . . 91
7.2.6 Interpreter . . . . . . . . . . . . . . . . . . . . . . . . . . 92
7.2.7 Scanner . . . . . . . . . . . . . . . . . . . . . . . . . . . 92



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7.3 Simulation Sub-System . . . . . . . . . . . . . . . . . . . . . . . 92
7.3.1 Simulation Sub-System Design . . . . . . . . . . . . . 93
7.3.2 Assumption . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
8 SYSTEM IMPLEMENTATION . . . . . . . . . . . . . . . . . . . . . 95
8.1 Class Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
8.1.1 Class Dependence Structure . . . . . . . . . . . . . . . 95
8.1.2 Class Specification . . . . . . . . . . . . . . . . . . . . . . 96
8.2 Concurrent Processes Management . . . . . . . . . . . . . 106
8.2.1 Concurrent Process Creation . . . . . . . . . . . . . 106
8.2.2 Communication . . . . . . . . . . . . . . . . . . . . . . . 107
8.2.3 CWB Invocation . . . . . . . . . . . . . . . . . . . . . . . 108
8.2.4 Main Program Linking with CWB . . . . . . . . . . . 108
8.3 Programming Languages . . . . . . . . . . . . . . . . . . . . 110
8.4 Environment . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
9 FURTHER WORK . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
9.1 Theory Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
9.2 Design Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
9.3 Implementation Work . . . . . . . . . . . . . . . . . . . . . . 112
10 CONCLUSION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
BIBLIOGRAPHY . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
VITA AUCTORIS . . . . . . . . . . . . . . . . . . . . . . . . . . . . 118



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 1 INTRODUCTION

Many design methodologies make use of graphical notation where software objects and relationships are represented using different symbols on a diagram. Rules exist governing how symbols should be used, how symbols should be linked and, in some cases, how symbols should be physically positioned in a diagram. One of most widely used methodologies which make extensive use of diagrammatic notations is the structured analysis and design technique (SADT).

SADT [6] deals with decomposing a system into modules. It uses dataflow diagrams (DFDs), entity-relationship diagrams (ERDs) and state transition diagrams (STDs), with the supplement of a data dictionary, to represent the static and dynamic properties of a system [37]. These diagrammatic notations provide not only techniques for system analyst but a structured approach to the development process. They are good for analyzing and structuring systems and are relatively easily understood by the customers. They also have the advantage of being well tried and understood and are used by the more conscientious developers of systems.

Among these three major diagrams, the dataflow diagram is the mostly common used one. DFD is a good tool for modelling data flows irrespective of physical and organizational boundaries and the medium of that flow. It provides a mechanism for ensuring a consistent hierarchical structure and is a useful analysis tool. Used sensibly it can provide an immediate and understandable model of the essential inputs, outputs and processes of the system. It is also a good design model, permitting the production of alternative information flows and providing a focus on discussion about the location of the human-computer interface. The



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

elements modelled — flows, processes, stores and terminators also lead to their physical equivalents.

As a user friendly and easy understanding graphical tool, dataflow diagram has been used in every phase in software life cycle. It plays active role in system design, system analysis, system maintenance, system reverse-engineering and software reuse. Its functional modeling features not only let system analyst to get good knowledge of a system behavior but also assist system designer to make a better logical structure of an object model.

## 1.1 The Role of DFD in Software Life Cycle

### 1.1.1 DFDs in Software Design

System analysis and specification are essential activities in any system development model. The languages used to describe specifications cover a broad range: from informal to formal, from operational to descriptive, from graphical to narrative. They usually include tables, diagrams, and other graphical notations which can convey information in a concise, rigorous, and readable way.

Though formal specification is very rigorous, precise and complete, in the real world, many companies are still reluctant or hesitant to use formal methods for system specification. Formal specification is not user friendly and hard to understand. It takes system designers a lot of time to transform formal specification into design model accurately. On the other hand, industries use narrative methods in system specification as less as possible to avoid ambiguity in the later stage of software development. Without doubt, graphical notations are commonly adopted by industries in software specification because they are intuitive, readable and user friendly. Dataflow diagram is one of the widely used



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

graphical tools in this early stage of software life cycle. It describes software requirements and provide an intuitive high-level picture of software functions and their decomposition into component parts.

Top-down approach is frequently used in the design stage of software life cycle. The modularity principle is of paramount importance in the design of software. The decomposition of a system into modules can be accomplished in several ways and in several steps. One might first do a decomposition in which the system is decomposed into higher-level module called subsystems. Relations among the subsystems are then defined, and the intended behavior of each subsystem is agreed upon by the designers. Next, each subsystems analyzed separately, and the procedure is iterated until reaching the point where the complexity of each component is sufficiently small that it can be implemented readily by a single person.

Dataflow diagrams provide a top-down, partitioned, graph-theoretic model for system design. Leveled DFDs present a good description of a system, its subsystems and relationship among the subsystems. A system/subsystem function decomposition has its corresponding components in module refinement and even in object-oriented module design. Each layer of module decomposition can be interpreted in corresponding level of DFDs. Leveled DFDs make system designers job easier and the design more readable and understandable as well.

### 1.1.2 DFDs in Software Maintenance

After a software is delivered, frequently required job is to modify the product to correct faults, or to improve performance to adapt the product to a changed environment. A delivered software may have some residual errors which could



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be present in any phase of software life-cycle such as requirements, specification, design, implementation, integration even maintenance, or could be any other types of errors. This is so called corrective maintenance that accounts for 20 percent of maintenance cost [16].

Most of the maintenance cost, namely over 50 percent, is spent on the second type of maintenance, perfective maintenance which involves changing the software to improve some of its qualities. Here, changes are made due to the need to modify the functions offered by the application, add new functions, improve the performance of the application, make it easier to use, etc. The request to carry out perfective maintenance may come directly from the software engineer, in order to improve the status of the product on the market, or they may come from the customer, to meet some new requirements.

The third reason for changing an application is adaptive maintenance which adjusts the application in order to react to changes in the environment in which the application operates. Adaptive maintenance can be a new release of the hardware or the operating system or a new database system. Thus this maintenance is not requested by a client; instead, it is externally imposed on the client.

Based on the activities described above, software maintenance can be divided into two categories: repairs and evolution, of which the second one claims most of maintenance work. Both these maintenance processes require system's maintainer to have, if not complete, good knowledge about the software product. However, the system's maintainer are usually not its designers, so they must expend many resources to examine and learn about the system. A well structured DFDs is very helpful for system maintainer to better understand the behavior of the system. They can modify existing functions or add new functions based on DFDs.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the worse condition which frequently happen, the only available documentation for a product that has to be maintained is the source code itself. In the course of developing software against a time deadline, the original specification and design documents are frequently not updated, and are consequently almost useless to the maintenance team. Other documentation such as the database manual or the operating manual may never have been written due to the priority of delivery time. Alternatively, continuing maintenance may have corrupted the original structure so much that it is no longer discernible. If no design documentation is available at all, product maintainer can draw themselves a DFDs based on system function test to acquire a whole picture of the system. Actually in industry, the most possible available design documents are architecture graphs or some flow charts similar to DFDs. It is not very difficult to create DFDs from these resources.

### 1.1.3 DFDs in Software Reuse

Software reuse is akin to software evolution. In software evolution, a product is modified for building a new version of the same product; in software reuse, a product is ready to be used, perhaps with minor changes, for building another product.

The candidates products for reuse can be all resources used and produced during the development of software [29]. Most frequently reused types of products are identified as:

1. data reuse, involving a standardization of data formats,
2. architectures reuse, which consists of standardizing a set of design and programming conventions, dealing with the logical organization of software,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. design reuse, for some common business applications and
4. program reuse, which deals with reusing executable code

Milis [19] has recommended a five-level hierarchy of reusable software development knowledge in which domain knowledge is represented explicitly:

1. environmental knowledge,
2. external knowledge,
3. functional architectures,
4. logical structures and
5. code fragments.

This classification corresponds somewhat to the software life cycle, where the last three levels map to the products of system design, detailed design, and coding. The first two (environmental and external) are typically used to derive a particular system's specifications from the user requirements.

The reuse of products of higher-level abstraction activities, such as architectures reuse or design reuse. gives greater leverage than code reuse. The higher-level reuse requires higher-level knowledge. One of the big problems of reuse is to acquire reusable assets. This activity involves various mixes of new developments and use of existing assets raw resources.

DFDs can be considered as the reuse of functional architectures, logical structures as well as design documents. This kind of reuse of high-level abstraction offers greater leverage. Reusing DFDs not only can help software error correction, improvement and maintenance but also can assist across project/program reference. A well designed and documented DFDs can either be pushed out from original creator to end user or be pulled out by new user from its original design.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 1.1.4 DFDs in Reverse Engineering

Reverse engineering encompasses a wide range of tasks related to understanding and modifying software systems. One of the dominant tasks is identifying the components of an existing software system and the relationships among them. Also important is creating high-level descriptions of various aspects of existing systems. The abstraction of a system could range from different phases of system life cycle to individual modules in the system, or it could be the design recovery of a software system.

What reverse engineering has done is to build up, more or less, a basis for maintenance, restructuring, reengineering and reuse of software, since successful executions of these processes rely on being able to recognize, comprehend, and manipulate design of a system. Even forward engineering, in the sense of system life cycle, involves a kind of reverse engineering.

Reverse engineering generally involves extracting design artifacts and building or synthesizing abstractions in a certain formality. These formalities are usually the methodologies used in software design. There are a couple of dozens of identified techniques used in software design. Each design methodology has its own notation (although these are often closely related) and its own set of rules defining how designs should be expressed using that notation. Figure 1.1.4–1 shows the reverse engineering concept.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

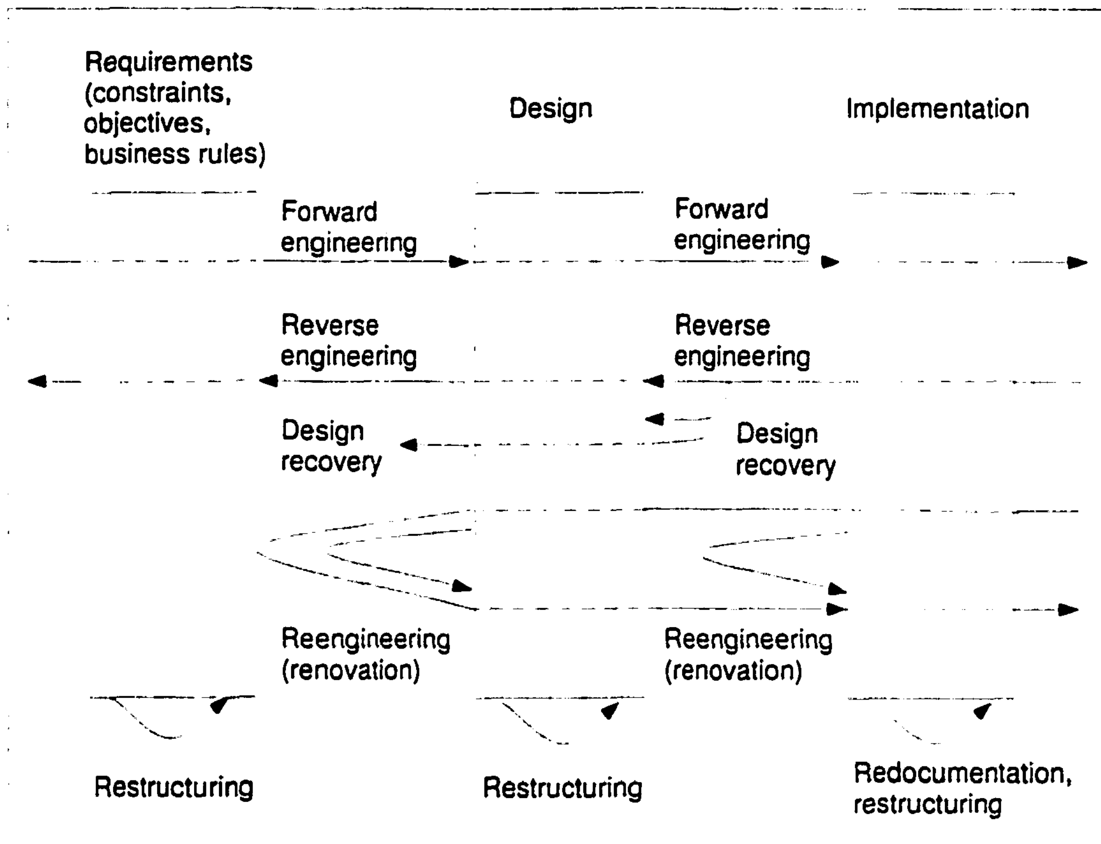



Figure 1.1.4-1 Relationship between terms. Reverse engineering and related processes are transformations between or within abstraction levels, represented here in terms of life-cycle phases

The term reverse engineering thus can be described as the process of analyzing a subject system to identify the systems's components and their interrelationships and create representations of the system in another form or at higher level of abstraction."[1]

Many of the models for high-level representation of traditional (sequential) software systems in literature tend to describe the system in terms of functional blocks and their interactions. These models are well defined as dataflow diagram



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

which is one of the most popular tools for the high-level representation of real time system.

Some of tools used in reverse engineering are to extract design properties of a system by reconstructing its dataflow diagrams either from executable code or from software specification documents. Some just reuse existing dataflow diagrams to help system analysts understand the behavior of systems. DFDs have been extensively used in software design and analysis for last one and half decades. Many large legacy systems were designed by using structured analysis and design technique with DFD-enhanced specifications. This is one reason that why some of reverse engineering methodologies focus on reusing or reconstructing dataflow diagrams. However, reverse engineering became popular both academically and commercially just in early 1990s. Thus DFD reuse and reconstruction in reverse engineering is still under development.

## 1.2 The Role of DFD in the Object-Oriented Paradigm

DFD is the most commonly used tool in functional modeling. A dataflow diagram shows the functional relationships of the values computed by a system, including input values, output values, and internal data stores. The processes in the functional model correspond to operations in the object model. Often there is a direct correspondence at each level of nesting. A top-level process corresponds to an operation on a complex object, and lower-level processes correspond to operations on more basic objects that are part of the complex object or that implement it. Sometimes one process corresponds to several operations, and sometimes one operation corresponds to several processes.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Processes in the functional model show objects that are related by function. One of the inputs to a process can be identified as the target object, with the rest being parameters to the operation. The target object is a *client* of the other objects (called suppliers) because it uses them in performing the operation. The target knows about the suppliers, but the suppliers do not necessarily know about the target. The target object class is dependent on the argument classes for its operations. The client-supplier relationship establishes implementation dependencies among classes; the clients are implemented in terms of, and are therefore dependent on, the supplier classes.

Actors are explicit objects in the object model. Data flows to or from actors represent operations on or by the objects. The dataflow values are the arguments or results of the operations. Because actors are *self-motivated* objects, the functional model is not sufficient to indicate when they act. The dynamic model for an actor object specifies when it acts.

Data stores are also objects in the object model, or at least fragments of objects, such as attributes. Each flow into a data store is an update operation. Each flow out of a data store is a query operation, with no side effects on the data store object. Data stores are passive objects that respond to queries and updates, so the dynamic model of the data store is irrelevant to its behavior. The dynamic model of the actors in a diagram is necessary to determine the order of operations [26].

## 1.3 Overview of the Approaches of DFD Processing

Because of its popularity, graphic view and intuitive meaning, dataflow diagrams have been considered to be good candidates of CASE support for



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

structured analysis and design since the mid 1980s. Those CASE tools either already available in market or still in the stage of research could be classified, in terms of their purposes, as editing tools, automatic generation tools, executing tools and reconstructing tools.

**Editing tools** are quite different from general graphic editing tools in use. General graphics tools use standard symbols — like rectangles, circles, lines and arrows ect. — to do basic graphic editing operations such as drawing, dragging, cutting, pasting moving and connecting. DFD editing tools only use DFD-specific graphic symbols, but not general drawing ones. They are usually much more intelligent than general tools. In addition to the basic drawing operations, DFD tools can check DFD syntax, detect duplicates, perform object search, automatically generate data flows, dynamically move or delete objects and related components, integrate DFD and data dictionary etc. Some advanced editing tools even can enforce diagramming rules, support concurrent DFD drawing, check consistency across diagrams and systematically replace objects with the diagrams at lower level in DFD hierarchical structure [28]. These DFD editing tools are usually so expensive that most system analysts can hardly afford to use them.

**Automatic generation tools** are created to save system designers from tedious and time-cost DFD drawing. All informations needed for drawing DFD are written in a structure representation using some descriptive language and then the representation is stored in a graphics database. The drawing subsystem access the database to retrieve flow information and parse it to generate the dataflow diagrams. By using automatic generation tools, a system designer/analyst can get a DFD automatically without any manual drawing. What they have to do is just using a descriptive language required by the automatic tool to write a structure



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

representation and inputting it into the database. Such kind of CASE tools are only used academically and still under improvement [24].

**Executing tools** provide a dynamic mechanism to simulate the behavior of a system semi-automatically. In such tools, a graphic dataflow diagram is converted to an executable specification in some formality — called an executable dataflow diagram — and then the executable DFD is read and interpreted by the executing system to generate a graphical dataflow diagram which can be used as a behavior simulation model for the target system. Various approaches are applied to form the executable specifications such as Petri net, token passing, set notation, pseudo-code description and flowmap [25] etc. All these approaches try to catch the semantics of the dataflow diagrams and control concurrence and dataflow sequences.

**Reconstructing tools** extract information from existing system documents and generate dataflow diagrams to help both system analysts and users to understand the system and to modify or update the system. Reconstructing DFD directly from code is a method used in reverse engineering. Sophisticated code analysis and transform analysis is involved in such reconstruction activities [33]. The other way to reconstruct a DFD is based on existing system design documents. A set of rules and definitions declared to transform design documents into graphical dataflow diagrams through a parser [7].

All these four kinds of CASE tools have different objectives, but share a common concept that a formal foundation should be created in order to draw dataflow diagrams automatically through a CASE tool. Each tool applied a specific methodology to set up a formal specification of dataflow diagrams which is either author defined or already existing, either mathematical or descriptive, either



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

process-oriented or structured. These experiences and various efforts stemming from the common idea indicate that a formal framework for DFD is likely to be the right route for automatic generation of and semantically analysis of dataflow diagrams.

Some of the CASE tools developed after late 1980s not only have the properties of first-generation CASE tools, which emphasizes remarkably sophisticated graphic-workstation user interfaces, but also catch some characteristics of second-generation CASE tools, which can provide methodology adaptation, documentation layout and intelligent diagraming support. Along with booming of reverse engineering and reengineering legacy system, more methodologies are proposed to support reuse of dataflow diagrams [18].

## 1.4 The Problems with Previous DFD Tools

DFD has been adapted to fit specific needs of different systems. Such adaptation includes changed notation, added notation and varied interpretations of some symbols. Here comes out a common issue for all the three ways using dataflow diagrams — "what kind of dataflow diagram is reconstructed ?". The second common issue for reconstructing dataflow diagrams is how to execute reconstruction. Drawing hierarchical dataflow diagrams manually for a large-scale system is extremely time-consuming and error prone if not impossible.

The solution for the first issue associated with the first two reconstruction methods is obvious, but it is not trivial if we reconstruct a DFD based on existing DFD with different notation or different interpretation of symbols. An intuitive solution for the second issue is naturally attributed to computer aided software engineering (CASE) [17]. CASE tools, especially DFD editing tools,



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

will definitely help drawing an eye-pleasing and standard DFD and save the system analyst from doing a tedious job. Regarding the use of CASE tools in reverse engineering, comes out another issue — "how a tool can guarantee that the reconstructed DFDs are logically correct and match the original design ?". Actually a DFD specification gives user some flexibility to modify it to fit particular needs of a specific system. The flexibility of DFD notation is one reason that leads this tool to be so popular. But the flexibility comes at a price — the lack of a formal basis of DFD concepts and notation hinder its use as a formal specification tool. The lack of formal framework is one reason that not a lot automated aids have been developed to support its use.

CASE tools that support DFD reuse must meet the following requirements:

- The reconstructed DFD should be syntactically error free.
- The DFD in different levels of hierarchy should keep consistency.
- Method rule checking should be embedded in the tool.
- The layout of DFDs should be aesthetically acceptable.
- The reconstructed DFD should be semantically equivalent to the original one.

Quite a few of CASE tools have been developed to support use or reuse of dataflow diagrams. Each of them can meet some of the above requirements to some extents, but not all of them.

The above described four major CASE tools for DFDs reuse all center on the syntactic aspect of DFDs. Though some of advanced those tools can support good-quality documentation, simple forms of consistency checking, bookkeeping even methodology adaptation and intelligent diagraming, they still can not interpret DFDs semantically.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Understanding the logical structure of DFDs is the key point to achieve use/reuse of DFDs in higher-level abstraction. CASE tools supporting intelligent use/reuse of DFDs rely on development of formal specifications for DFDs. Formal specifications can describe dataflow diagrams either syntactically or both syntactically and semantically. Formal specifications for DFD can not only help generate precise and consistent diagrams but also give a meaningful interpretation and help systems analysts understand the behavior of the described system. The semantic specifications for DFD is also known as executable specifications that allows the drawn dataflow diagrams to be executed to simulate the behavior of the underlined system.

## 1.5 The Organization of the Thesis

The major remainder of this thesis paper is organized as nine sections. Section two gives a brief description of dataflow diagrams, its symbols, terminology, notation and construction. Section three evaluates the CASE tools for constructing dataflow diagrams in structured analysis and design, and investigates the various methodologies used in different tools. Section four focus on semantic representation of dataflow diagrams with introduction to CCS. Section five proposes a semantic driven dataflow diagram processor. Section six develops the specification for the proposed system. Section seven analyzes the system design issues. Section eight describes the implementation of a simulation sub-system. Section nine is reserved for further work. The last section gives a brief conclusion.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 2 DATA FLOW DIAGRAMS

## 2.1 Overview

In software engineering, a system development is usually processed as a life-cycle model, especially for large-scale software design. The early phases of a software production process deal with more abstraction aspect of the system which is generally represented by various specifications ranging from requirements specification to design specification. A specification is a statement of an agreement between a producer of a system and a consumer of the system at any stage of the life-cycle model of the system. It can be used for different purposes such as a statement of user needs, a statement of the requirements for the implementation, or a reference point during product maintenance.

Software specification may take any form of representations which can be formal or informal, and also can be operational or descriptive. While formal specifications can be presented by an algebraic specification language or a logic specification language such as Z notation, informal specifications are written in a natural language or a language associated with some figures, tables, diagrams and other notations to help understanding. Descriptive specifications try to state the desired properties of the system in a purely declarative fashion like entity-relationship diagrams. By contrast, operational specifications relate the intended system by describing the desired behavior, usually by providing a model of the system, i.e., an abstract device that in some way can simulate its behavior. A dataflow diagram is a good example of operational specification.

What can be used as specifications in software development must meet certain requirements. The first quality required of specifications is that they should be



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

clear, unambiguous, and understandable. The second is consistency and the third is complete.

Of the various specifications used in software engineering, the most often used, acceptable, easy understand ones are diagrams. There are three major graphical modeling tools of structured analysis and design technique (SADT): data flow diagram (DFD), entity-relationship diagrams (ERD) and state-transition diagrams (STD). While STD highlights time-dependent behavior of a system and ERD presents a data model of a system, DFD models the functions performed by a system. Dataflow diagram is also known as some other terms like: Bubble chart, Bubble diagram, Process model, Work flow diagram, function model.

The dataflow diagram is perhaps the most commonly used systems-modeling tool, particularly for a systems in which the functions of the system are of paramount importance and more complex than the data that the system manipulates. DFDs were first used in the software engineering field as a notation for studying systems design issues. In turn, the notation had been borrowed from earlier papers on graph theory, and it continues to be used as a convenient notation by software engineers who are responsible for direct implementation of the models of user requirements.

Since DeMarco [6], who is one of the first those who describe DFDs in a systematic, instructive way, and Gane and Sarson [12], who also use DFDs as a major tool in describing system analysis and design, published their books: *Structured Analysis and System Specification* and *Structured Systems Analysis* respectively in 1979, DFDs had been extensively used as a graphic tool in system analysis and design. Different notations and conventions were adopted to meet special needs in specific system development. In 1989, Yourdon [37] summarized



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the experiences of ten-year use of DFD and proposed a set of notations and rules concerning DFDs which is usually called *Yourdon dataflow diagrams.*

Besides their characteristics of hierarchical structure and more complete notations compared to other graphical tools of structured analysis, Yourdon dataflow diagrams have all the three major qualities required of specifications. It is also one of the most popular DFD conventions accepted in software industry. My research in DFD will be based on Yourdon DFD model.

## 2.2 The Components of DFDs

A dataflow diagram consists of a number of graphical symbols, which are circles, rectangles and lines. Circles, rectangles and two parallel lines are connected by labelled, directed lines which represent data "flowing" through the system, with each one using some or all of its input data to produce its output. There are four major components of Yourdon dataflow diagrams, which are *process* (*transformation*), *dataflow*, *store* and *terminator* as shown in Figure 2.2–1. There are also some minor components, which are *control transformation*, *control flow* and *event store*. Since they are not in my interest, they are not introduced [35].



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

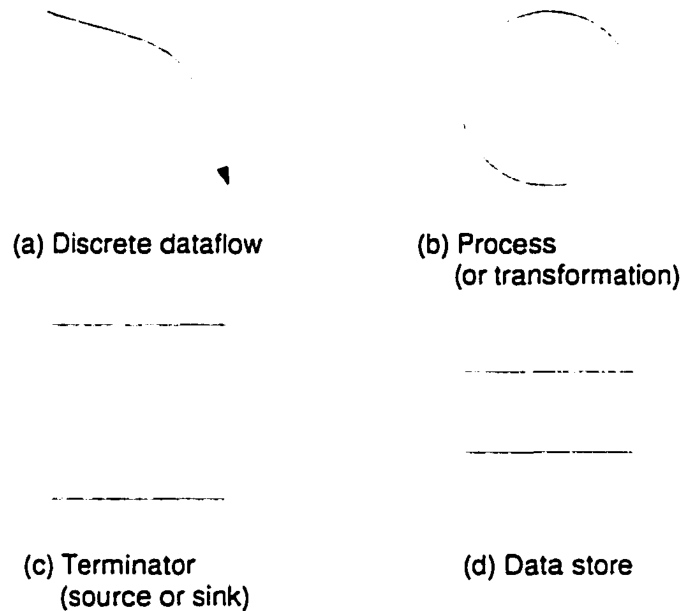



Figure 2.2-1 The symbols of Yourdon dataflow diagrams

### 2.2.1 The Process

The first component of the DFD is known as a process. Common synonyms are a bubble, a function, or a transformation. The process shows a part of the system that transforms inputs into outputs. It shows how one or more inputs are changed into outputs. The process is represented graphically as a circle, as shown in Figure 2.2–1(b).

### 2.2.2 The flow

A flow is represented graphically by an arrow into or out of a process; an example of flow is shown in Figure2.2.2–1. The flow is used to describe the movement of chunks, or packets of information from one part of the system



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to another part. Thus, the flows represent data in motion, whereas the stores represent data at rest.

In order to obviate the use of the logic "AND" and "OR" or the operator * and ⊕, Yourdon gives some rules of composite flows illustrated in Figure 2.2.2–1 and Figure 2.2.2–2. Consider Figure 2.2.2–1. The diagram (a) shows the flow X going to two processes — A and B; diagram (b) shows X diverging into two flows — z and y that go to A and B respectively; diagram (c) shows the flows d1 and d2 converging to one flow DD for T needs both d1 and d2 to process; diagram (d) shows the flows d1 and d2 going to T separately for T needs only one of them to process.

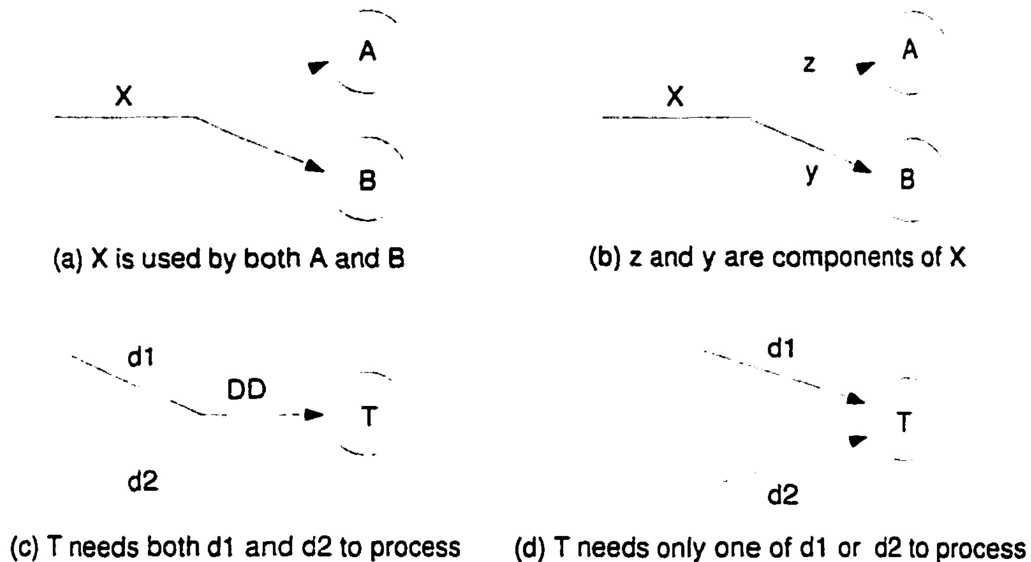



Figure 2.2.2-1 The rules for composite flows as input data

In Figure 2.2.2–2, the diagram (a) implies that p and q produced by D and E respectively are part of the composite dataflow R; diagram (b) shows two



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processes — D and E — both producing the dataflow R, but not at the same time: under some conditions, D will produce R, under others E will produce R; diagram (c) shows that the flow OO produced by T2 is composed of O1 and O2; diagram (d) depicts that T2 alternatively produce O1 or O2, but not at the same time.

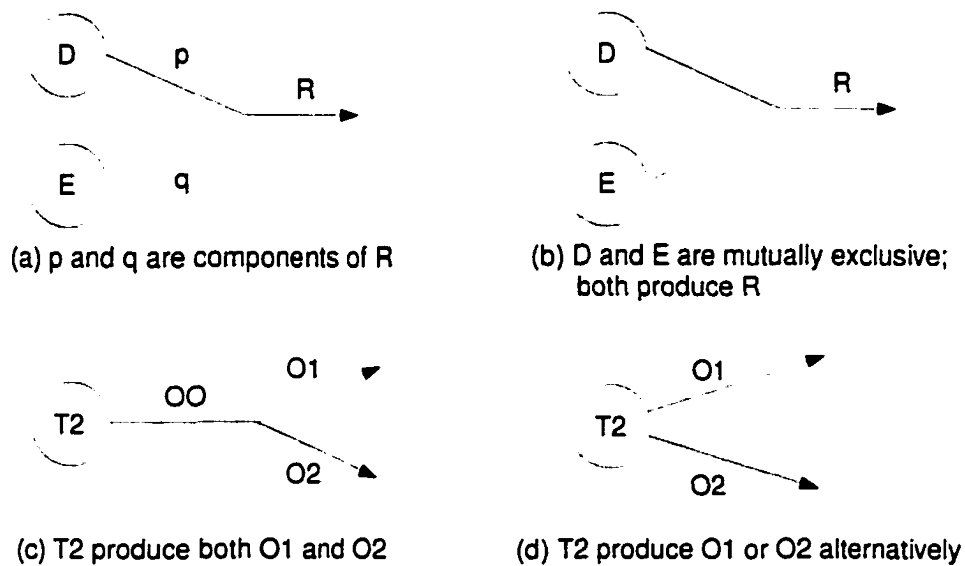



Figure 2.2.2-2 The rules for composite flows as output data

### 2.2.3 The Store

The store is used to model a collection of data packets at rest. The notation for a store is two parallel lines, as shown in Figure 2.2–1 (d). Store can be used as a necessary time-delayed storage area between two processes that occur at different times, as a convenient temporary repository of data between two implementations or as an independent storage from which data is extracted or into which data is sent.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Data can not flow directly from a store to a terminator, or from a terminator to a store; in either case a process is needed to process the data. In most cases, the flows will be labeled, but many systems analysts do not bother labeling the flow if an entire instance of a packet flows into or out of the store.

Store is passive, data will not travel from the store along the flow unless a process explicitly asks for them. While store is not changed when a packet of information moves from the store along the flow, a flow to a store is often described as a write, an update, or possibly a delete.

### 2.2.4 The Terminator

Terminator is graphically expressed as a rectangle as shown in Figure 2.2–1(c). Typically, a terminator is an outside agency or another system. It represents external entities with which the system communicates. The flows connecting the terminators to various processes or stores in a system represent the *interface* between the system and the outside world. The terminator from which data flows come out is a source of the system and the terminator to which data flow goes in is a sink of the system.

## 2.3 Constructing DFDs

There are few hard-and-fast rules regarding the use of dataflow diagrams. Most of systems analysts create dataflow diagrams by experiences and knowledge of structured design. However, some conventions are widely accepted through the past two decades of DFDs development such naming, numbering, proper number of processes in one diagram and etc. Figure 2.3–1 shows an example of a cooking system represented by DFDs. Point A is a merging spot of four data flows: two



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of them — *Diced carrots* and *Fried onions* come from processes *Prepare carrots* and *Fry onions* respectively; the other two — *Water* and *Seasoning* come from source terminators *Tap* and *Spice rack* respectively. Process Cook ingredients needs all four flows to start transformation.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

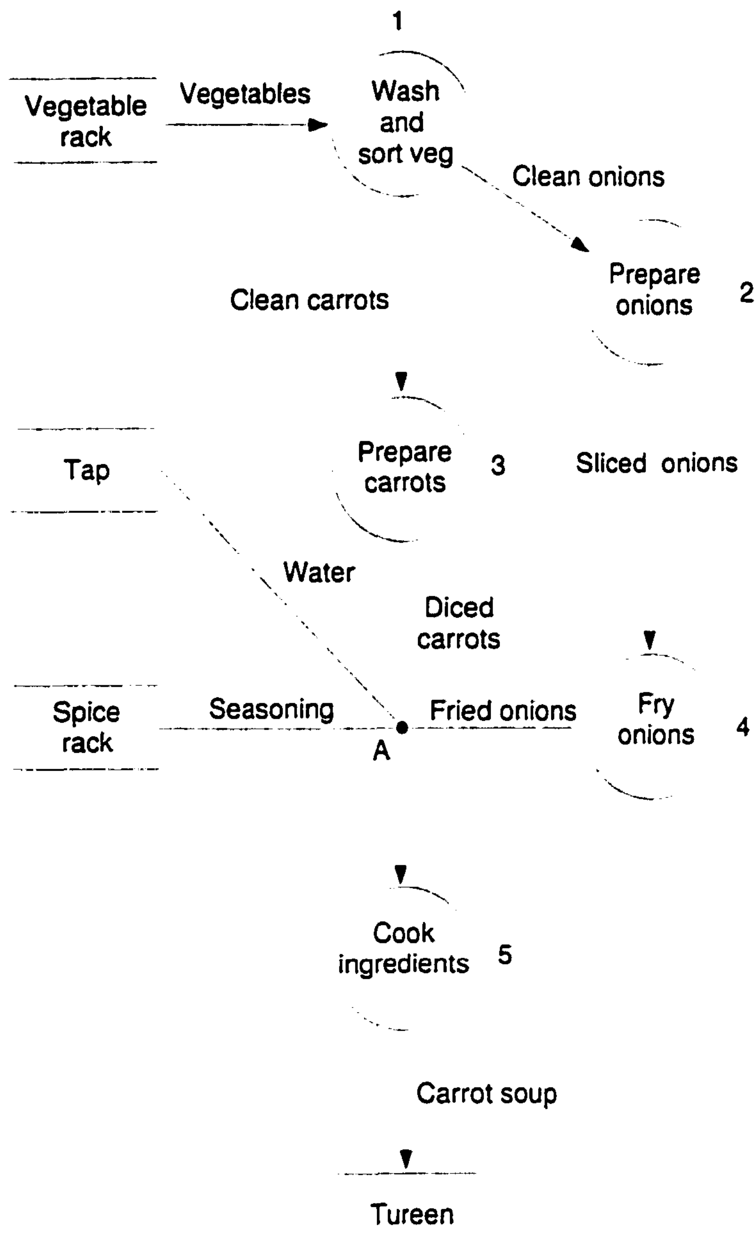



Figure 2.3-1 A cooking system represented by DFD



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 2.3.1 Leveling and Balancing

The most often used method of creating DFDs is to construct DFDs of a system in a series of *levels* so that each level provides successively more detail about a portion of the level above it. This strategy is also known as *functional decomposition* or *dataflow diagrams refinement.* Theoretically, it is an application of the concept of hierarchy which is very old, but very simple abstract idea.

Functional decomposition begins at the boundary between the software system and its environment. The top-level DFD is a so-called *context diagram* and constitutes the root of a hierarchy of functions required of the system. A context diagram is a dataflow diagram which contains a single transformation that represents the entire system and the major sources of data and destinations for data in the environment. (Indeed sources and sinks usually only appear in the context diagram.) The function of main transformation of a context diagram is then decomposed and the circle which represents it is refined into a diagram whose transformations are further refined, and so on until a *functional primitive* is constructed. Functional primitive is a transformation which cannot be refined any further and can occurs at any level of abstraction. Repeated decomposition and transformation refinement results in a hierarchy of dataflow diagrams. Such a hierarchy is called a *levelled set* by DeMarco [6].

Figure 2.3.1–1 gives an example of levelled dataflow diagrams. On the top of the DFDs is a context diagram within which the only process is labelled by a noun rather than a verb describing a transformation. Usually, the name of the process in the context diagram is the same as the name of the system such as *XYZ system* in the figure.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

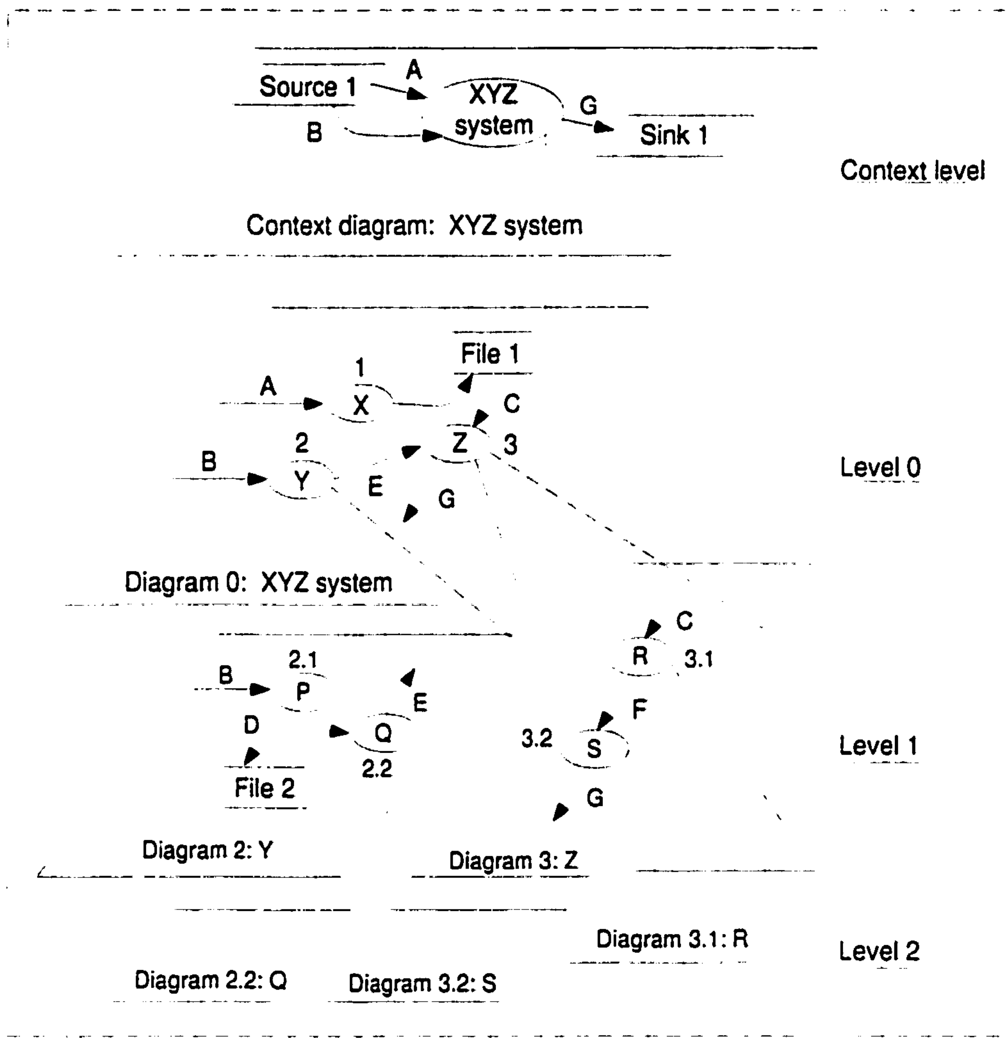



Figure 2.3..1-1: Functional decomposition of DFDs

The process of the context diagram is decomposed down to the next level of the DFDs which represents the highest-level view of the major functions within the system, as well as the major interfaces between those functions. The level immediately beneath the context diagram is usually numbered 0 and the diagram at this level is also numbered as 0 such as *Diagram 0: XYZ system* in the figure.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

All the transformations are numbered to be identified at this level and lower levels such as, in the figure, transformation *X* is numbered as *1*, *Y* as *2* and *Z* as *3*.

Process *Y* in *Diagram 0* is further decomposed into a more detailed diagram with the same number and name as *Y* has. This diagram is *Diagram 2: Y* in the figure. All the bubbles in *Diagram 2: Y* are associated with the bubble 2 of upper-level DFD and are numbered 2.1 and 2.2. At *Level 1*, another diagram *Diagram 3: Z* is constructed by decomposing process *Z* in *Diagram 0*. The corresponding bubbles associated with the bubble 3 of upper-level DFD are numbered 3.1 and 3.2. Then comes *Level 2* consisting of *Diagram 2.2: Q*, *Diagram 3.1: R* and *Diagram 3.2: S* which are functional primitives since no more decomposition beyond this level.

The use of the primitive concept does constitute a convenient stopping rule for the work in analysis. Some processes are simple enough that it makes no sense to require breaking them down to the same level of detail as others that are more complex. To determine whether a process is simple enough to be considered as a primitive, two checkpoints are usually applied by experience. If a reasonable process specification for a bubble cannot be written in about one page, then it probably is too complex and should be partitioned into a lower level DFD. The other idea is to write a reasonable pseudo-code for a process. If the pseudo-code is more than 50 to 100 lines long, the process should be refined to a lower level.

While leveling a DFD, balancing it is as well important. The original consideration behind the balance is that complexity between different diagrams at any level of a DFD shouldn't have much discrepancy. In order to make a DFD easily readable and understandable, each diagram shouldn't have more than half a dozen processes and related stores, flows, and terminators. That also means



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

that a DFD which contains reasonable size of symbols and characters should fit comfortably onto a standard letter-size sheet of paper.

### 2.3.2 Repartitioning

Although it is recommended in general that leveling be used to decompose systems top-down, top-down is not always the best approach. In fact, the top-down strategy does not work as well as the bottom-up strategy. Experience with Structured Analysis (and other methods) has shown that most analysis is actually conducted in a bottom-up fashion, with a top-down scheme being used to organize those results [23].

Upward repartitioning is just decomposition in reverse — synthesis rather than analysis. It involves developing a detailed model based on whatever information has been acquired and examining the model to determine whether or not there are any bubbles or processes which are related by virtue of the nature of the tasks they perform. In the top-down approach one basically imposes one's own view of how the system ought to be structured. In the bottom-up approach, to a much greater extent, the system is telling us just what it is structured like. During the design of a system, both repartition upward and decomposition downward are used to achieve a uniform level of detail.

### 2.3.3 Evaluating and Improving DFD

While a number of rules and guidelines that help ensure the dataflow diagram is consistent with the other system models — the entity-relationship diagram, the state-transition diagram, the data dictionary, and the process specification, there are some guidelines that help dataflow diagram itself consistent.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

First of all, infinite sinks and spontaneous generation bubbles of a system must be avoided. Bubbles definitely have both input and output flows. The bubble which has only input but no output or the bubble which has only output but no input will result in logical error within the system. Secondly, unlabelled flows and unlabelled processes in a system should be given names before they connect other elements in the system. Because such unlabelled symbols may cause several unrelated elementary data items to be arbitrarily packaged together or cause dataflow diagram to be degraded to a disguised flowchart. Finally, read-only or write-only stores within a system are not allowed. A typical store should have both inputs and outputs. The only exception to this guideline is the external store, a store that serves as an interface between the system and some external terminator.

In order for a DFD to be technically correct and acceptable to users, it should have been drawn, redrawn, and redrawn again, often as many as ten times or more before it is passed to a user [6]. This may seem like a lot of work, but it is well worth the effort to develop an accurate, consistent, esthetically pleasing model of the requirements of a system. Consequently, demand for automated tools for DFD arises as well as reuse of DFD which will be examined in the next section.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 3 CASE TOOLS FOR DFDS

## 3.1 The Role of CASE Tools in Software Development

Just as CAD/CAM technology has helped revolutionize various engineering disciplines over the past 35 years, so CASE (computer-aided software engineering) technology is helping revolutionize the software industry. At present, some professional programmers and system analysts are equipped with some CASE tools but many are not. Thousands of CASE tools, which support different activities in the software process, are commercially available.

CASE tools are currently being used in all the phases of software engineering process. CASE tools that help software developers during the earlier phases of the process, namely the requirements, specification, planning, and design phases, are sometimes called *upperCASE* or *front-end* tools, whereas those that assist with implementation, integration. and maintenance are termed *lowerCASE* or *back-end* tools [27] Both front-end and back-end tools are recognized as *activity-oriented* tools because they are base on process activities. Another classification scheme based on the functionality of the tools rather than the activity which the tools support is called *function-oriented* [22].

An important part of supporting the software life cycle is supporting the methodologies that structure the process steps within the life cycle. A CASE workbench supports the use of structured methodologies by automating the production of the documentation required by the methodology and guiding the user in the correct use of the methodology. CASE technology that emphasizes the early stages of the life cycle comes from recognizing analysis and design as the



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

most critical life cycle phases. These CASE tools are known as *systems analysis and design workbenches* [10].

Specification errors can be very expensive if they are not detected and corrected in the early phases of the software development. Correcting a specification error during the maintenance phase is a lot more expensive than if it had been corrected during the analysis phase. The completeness and correctness of the system specification affect the success of the entire software development effort. The specification is the basis for project schedules and assignments, test plans, user documentation, and program design. Poorly-understood system requirements cause software failures.

Design errors often dominate software projects in terms of their number and their cost to correct, especially when not detected early. In large projects, design errors often exceed coding errors and are more costly than coding errors to correct as well. More care given to design means lower-cost, more reliable systems. A system design is the blueprint for system implementation. If the blueprint does not exist or if it is incorrect, the produced system is probably poorly organized, poorly documented, and a nightmare to maintain.

Systems analysis and design workbenches first emerged about ten years ago. These workbench tools are primarily concerned with the effective development of the models of a system that is to be computerized, they help the systems analyst construct graphical diagrams that enable the end user to understand what the system will do for him. The workbenches also help the analyst and designer ensure that the model is complete, accurate, and consistent, so the errors discovered downstream in the programming phase will be only programming errors, and not a reflection of ongoing misunderstanding between the end user and the systems



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

analyst. And, finally, the workbenches may assist the programmer in translating the model into a working program. In the future, we may expect the workbenches to completely automate this process.

## 3.2 Important Features in CASE tools

The workbenches for systems analysts and designers have to provide the following features to be of significant use in the development of complex system:

- Graphics support for multiple types of models.
- Error-checking features to ensure model accuracy.
- Cross-checking of different models.
- Additional software engineering support.

### 3.2.1 Graphics Support

Structured analysis models rely on various forms of information: text, data dictionaries, and graphical diagrams. Text and data dictionaries can be automated using word-processing systems and conventional mainframe computers: but graphics support is not as popular as text does. An analyst workbench should allow the systems analyst to compose, revise, and store diagrams such as dataflow diagrams, structure charts, flowcharts, entity-relationship diagrams and state-transition diagrams.

### 3.2.2 Error-Checking Features

An analyst workbench must examine the model created by the systems analyst or designer to ensure that it is complete and internally consistent. For example, a dataflow diagram created by a CASE tool must complies with all the rules



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

described in Section 2 and the names assigned to each process must be unique. The error-checking also should be extended to different levels of modeling to make sure that the input and output of a process match those of corresponding diagrams at both the lower and higher levels.

### 3.2.3 Cross-Checking of Different Models

The most important feature of an analyst/designer workbench is its ability to cross-checking the consistency of several different types of models of a system. This kind of checking can be classified into two aspects: cross-checking different models in one phase of a project and cross-checking different models at different phases of project.

In the system-analysis phase of a project, for example, the primary objective is to determine what the user wants from the system, with little or no concern to implementation of those requirements. For this purpose, DFDs can be used to highlight the division of those requirements into separate functions and the interface between the functions, a data dictionary is needed to maintain a definition of all the data elements in the system and some form of textual description to define the formal business policy. All these models must be consistent with one another. If the DFD refers to a data element that is not in the data dictionary, something is wrong; if the data dictionary defines data elements that do not appear in DFD model, something is also wrong. It is not hard to imagine how tedious and errorprone it is if this cross-checking is done manually.

Complementary to the consistency checking between models in one phase of project, it is as well important to compare the models developed during different phases. For instance, the models developed during the analysis phase should be



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

compared with the models developed during the design phase. This comparison should demonstrate a one-to-one correspondence between the two. Every requirement described in the analysis model should be represented somewhere in the design model, and every feature described in the design model should correspond to a requirement described somewhere in the analysis model. The most common problem, of course, is that a requirement in the analysis model gets dropped and doesn't show up anywhere in the design model. This is particularly common when the systems analysis model is developed by one group of people, and the design model is developed by a separated group.

### 3.2.4 Additional Software Engineering Support

Other supports can be classified as many aspects ranging from software life cycle to structured methodology. They may include CASE tools support networks for project-wide use, software engineering methodology, document control, project management facilities, early checking for excessive complexity, computer-assisted proof of correctness, automated testing and simulation and reuse of software components on any phase of the software engineering process.

Many of the features described above exist in the analyst designer workbenches in the market today. Some of the features are implemented in a somewhat primitive form, especially for the additional features, but the products are being improved on almost a daily basis. The CASE tools for other features, such as reuse of software components and reuse of software documents, are still under development.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 3.3 Current CASE Tools for DFDs

CASE tools for DFD have been developed for the past ten years. The achievement is ranging from the design workbench for an automatic arrangement of symbols in a DFD to computer-assisted reconstructing a DFD in a legacy system. These systems analysis and design workbenches all focus on facilitating systems analysts or designers to create, edit, check or reconstruct DFD automatically other than manually. The following part of this section will describe the achievement of CASE DFD support so far.

### 3.3.1 DFD Editor and Processor

Since T. DeMarco's *Structured Analysis* [6], as well as C. Gane and T. Sarson's book [12], was published in 1979, dataflow diagram have become the most popular notational tool of structured systems. But manually drawing DFD is tedious, error prone, terribly burdensome to do any checking, very time-consuming and very expensive The layout algorithm for DFD described above only can improve the view of an existing diagram according to aesthetics, lots of work still have to be done manually by the systems designer. Editing tools supporting dataflow methodologies are badly needed by systems analysts and designers.

The requirements of an intelligent DFD tool are described as:

It should enforce consistent definition of each element in the diagrams and detect duplicates to maintain the integrity and consistency of the data dictionary.

It should have the intelligence to generate optimal routes for dataflows so that the diagrams are eye-pleasing for the analysts to understand easily.

It must allow dynamic modification of diagrams by moving or deleting objects and their related components with minimum effort from the analysts.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It should encourage partitioning by allowing the child diagrams to be concurrently edited with their parent diagrams in a user-friendly manner.

It should also support systematic replacement of any element [28].

The following editing tools are used in assisting software design and analysis. They can help establish an interactive development environment and provide graphic editors to support for several widely used analysis and design methods, including structured systems analysis and structured design. Although these tools were helpful during design process, they have not been widely accepted. There are several reasons. First of all, many programmers are skeptical about disciplined software-development methodologies and stick to the way they used to do [38]. Secondly, these tools are expensive compared to general graphic editing tools [28]. Thirdly, many tool users prefer general graphic tools to specific ones [5]. However, the concepts of disciplined software development and structured design are especially valuable to the design and analysis of large-scale system. Along with the progress of the CASE tools, they will be more and more widely used as design tools [17].

**Macintosh Anatool** Anatool has three major components: a dataflow-diagram editor, a data dictionary, and standard specifications and utilities. The dataflow-diagram editor automatically numbers each diagram in the hierarchy structure and each process bubble in the diagram [36].

The first step in creating a dataflow diagram is to create level 0, the highest level in the diagram hierarchy. The left side of dataflow-diagram editor window is a control palette of nine drawing tools. The top box represents a terminator (source or sink) outside the system's scope, such as a user. The second represents



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a process that must be performed by the system. The bottom seven drawing tools are for data stores, dataflows (one- and two-way), word processing, and hand-scrolling, zooming out, and selecting components.

Process, external entities, and data stores are placed into a diagram by selecting components with a mouse, dragging them out of the control palette, and placing them at the desired location. Dataflows are placed in the window by selecting the desired flow and clicking on the source and destination positions. Anatool determines automatically which side of the source and destination components to draw the dataflow from or to. It also determines how to draw the dataflow.

Process bubbles, dataflows, terminators, and data stores can be repositioned by dragging them around the window. When an entity is moved, all flow connected to it also move. Anatool has a sophisticated way to reroute the flow and redraw the whole diagram. The number of process bubbles per diagram is limited. This means that the user of this tool has to decompose complicated processes to keep each level manageable and readable. The size of process bubbles, data stores, and terminators are fixed. Everyone of them must be labelled with no more than 30 characters. The labels assigned to data stores and dataflows are automatically entered into the data dictionary, but the labels of processes and terminators are not.

Clicking on a process bubble will refine it into lower level of dataflow diagram. When it creates a child diagram for an existing process, Anatool automatically puts the external sources and data stores form the upper level into the child's diagram along with *bridges* which represent all the processes that were connected to the partitioned process from the upper level. Establishing connections between different levels is a nice feature and helps keep the diagram consistent, but it is hard to remember all the informations in different levels since



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Anatool doesn't allow multiple windows.

**Macintosh MacBubbles** MacBubbles Version 1.9.2 consists of two programs: MacBubbles and the MacBubbles data dictionary [21]. The first is a graphics-based editor for creating Yourdon/DeMarco-style dataflow diagram and mini-specifications, while the second is a dictionary-maintenance utility [39][6].

MacBubbles uses a MacDraw-style interface, with a palette of shape tools on left side. The shape tools consists of both basic symbols of dataflow diagram and extended control symbols. The way the MacBubbles creates a DFD is similar to that the Anatool dose. One of the former's advantages over the latter's is that MacBubbles supports very flexible flow lines that can arc and curve as desired. Terminators, process bubbles can be enlarged or reduced. Data store object can be rotated on the screen so that dataflows take a more direct path to and from the data store. The resulting diagrams are more visually pleasing. Like Anatool, MacBubbles constructs dataflow diagrams hierarchically but doesn't allow multiple windows

**AUTO-DFD** While it has all the features that both Anatool and MacBubbles have, AUTO-DFD is much more "intelligent" than they are. AUTO-DFD can integrate DFD and data dictionary, detect duplicates, enforce DFD diagramming rules, perform object search, automatically generate dataflows, dynamically move or delete objects and related components, support multi-windowing to edit diagrams of different levels concurrently, check on the integrity of all entities of the dataflow diagrams and on the balance of input and output flows between a process and its child diagram, systematically replace objects with their child diagrams, compress diagrams, find the optimal dataflow path between two entities



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of a dataflow diagram, and provide on-line help [28]. It seems that AUTO-DFD could meet all the requirements of an intelligent DFD tool.

The design of AUTO-DFD is object-oriented and aims to provide a completely visual environment for analysts to model the information system by manipulating icons on screen. The architecture of AUTO-DFD is shown in Figure 3.3.1–1. The graphical interface for editing, as shown in Figure 3.3.1–2, is a typical editing window for DFD in late 1980s and early 1990s [28]. Anatool and MacBubbles all have the similar iconic interfaces.

A routing algorithm that relies on heuristics has been devised for AUTO-DFD to find a visually acceptable dataflow path between two objects. To find a qualified path, the algorithm considers routes with not more than three turning points. In each case, it will give priority to routes with minimum crossings and then shortest distance.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

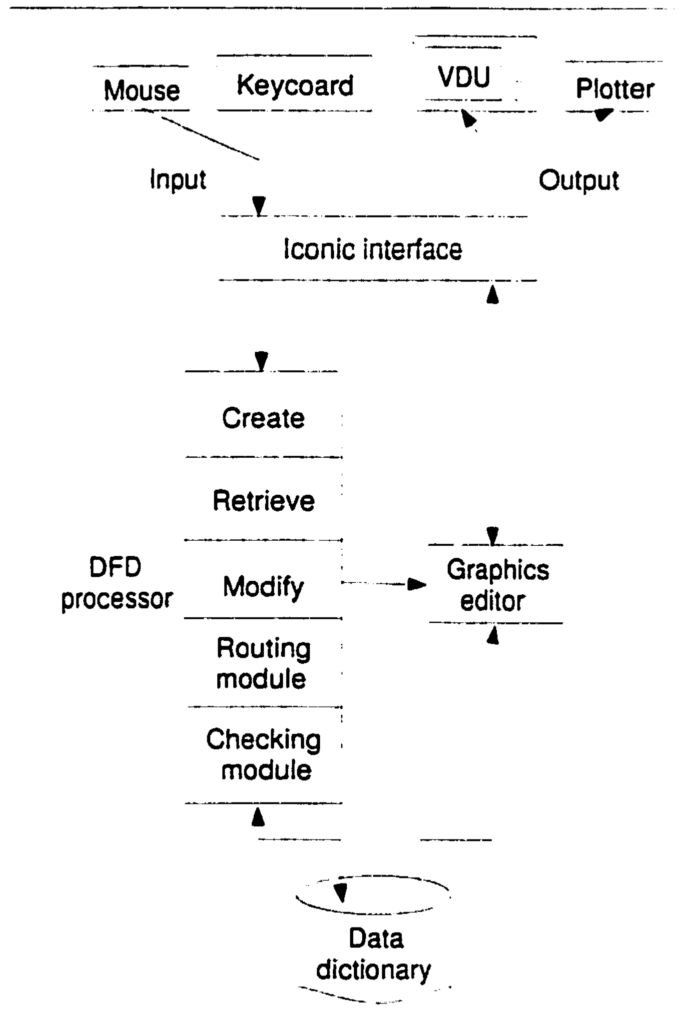



Figure 3.3.1-1 The architecture of AUTO-DFD



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

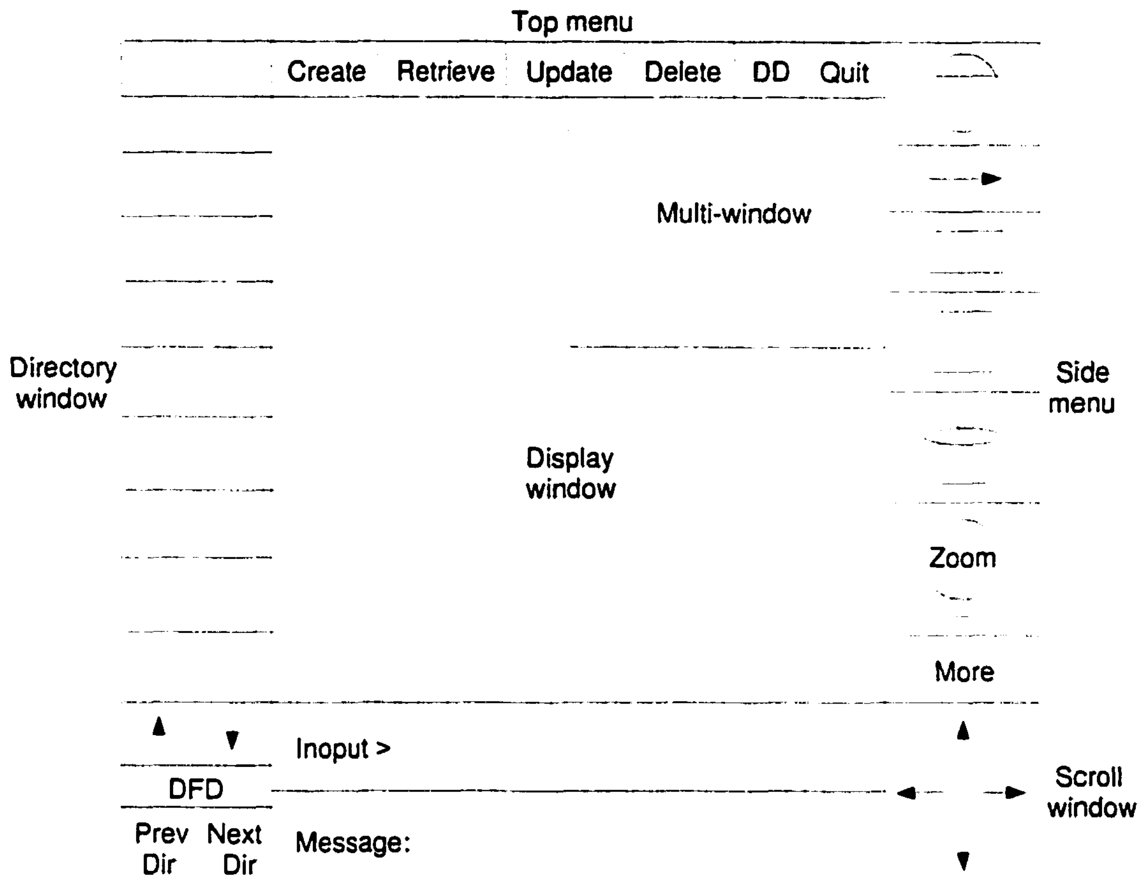



Figure 3.3.1-2 The graphic interface

**FLEDGED** FLEDGED belongs to the second-generation of CASE tools. The first-generation CASE products have emphasized remarkably sophisticated graphics-workstation user interfaces. They help users develop systems-analysis diagrams and detailed specifications but not automatically, such tools like Anatool and MacBubbles described above. The second-generation tools are characterized by the following features:

- Support various analysis techniques the analysts want.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Produce hard-copy documentation automatically along with CASE tool.
- Automate the production of systems-analysis diagrams [5].

While AUTO-DFD has addressed the routing problem of automatic layout, FLEDGED has touched the tool-tailoring problem. FLEDGED is a flexible editing tool which allows users to define a graphical symbol to their taste for each type of dataflow-diagram element, to define their own set of formation rules, to define their own set of editing operators, and enforces formation rules automatically during performing editing operators [15].

FLEDGED has a symbol library, which contains all the possible symbols of various versions of dataflow diagrams, from which a user can choose one pair of shape type and drawing style for each process type, terminator type, and store type. The formation rules are formulated as logical rules, logical relations on structural functions. Every time when a formation rule has been successfully defined by a user. it is stored in a rule base and then automatically translated into checking procedures. ERA (entity-relationship attribute) framework with a shell of primitives called *structural functions* and *structural operators* is enclosed in the tool to support the definition of formation rules and editing operators, and to support the enforcement of formation rules during editing operations. Structural operators are primitive operators that change the structural details of the intended ERA system model. Editing operators are defined as procedural compositions of structural operators. FLEDGED provides two ways to check a rule: explicit invocation which is prompted by command *check rule*, and automatic enforcement which is in effect with command *enforce rule*.

**Method rule checking in DFD editing systems** One of the important issue in



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

editing tools is to ensure that the edited diagrams are correct and comply with all the DFD construction rules. How to keep consistency in DFD editing depends on the design of DFD tools. Tools available for software design diagram editing can be categorized in three principal ways. *method-specific* or *configurable*, *syntax-driven* or *permissive*, and *stand-alone* or *integrated* [34].

The tools restricted to one or a group of methods are considered as method-specific such as Anatool or MacBubbles, those that allow tool builders to specify their own methods or local variations on existing methods are configurable such as FLEDGED. These tools must contain some rule-checking mechanisms within the editing system.

A syntax-driven approach maintains a correct diagram at all times, forcing the user into a rigid interaction style. A permissive approach allows diagrams to go through incomplete or inconsistent states, and there is a choice between interactive and off-line checking.

Some tools allow the user to draw diagrams, store them and edit them, but further manipulation of the stored diagram representation is left to the user. These tools are considered as stand-alone. Integrated tools allow other types of tools, such as code generators, to manipulate the output from the design editor.

For method-specific tools, automatic method rule checking should be incorporated to support the production of designs expressed in method-specific diagrammatic notations. For configurable tools, the method-rule checking could be tailored to any notation using a method description language and a graphical tool to define the vocabulary of the notation. Both syntactic and semantic rules are expressed in the method description language and are checked, interactively, dur-



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ing an editing session. Such a method-rule checking system is investigated by Ray Welland [34].

The alternative strategy, adopted by M-J Chen, is a preventive approach to structural analysis [4]. The approach associates structural checking with editing operators so that editing operators that will introduce structural errors into DFDs are inhibited. If this strategy is described as pre-checking before editing, then Welland's method is spontaneous checking. Of course not all of the structural errors can be prevented, a decision on which structural errors can or should be prevented must be made. The decision is based on consideration of two factors: the characteristics of structural errors and the construction methods for DFDs. M-J Chen classified a set of assumptions and restrictions based on a combination of Yourdon's, DeMarco's, Ward and Mellor's convention of dataflow diagrams and a set of formation rules that follows these assumptions and restrictions. The defined formation rules support system analysis methods which include functional decomposition and editing operators, and support event partitioning as well. These formation rules are described as logical languages that are used as assertions to ensure consistent DFD editing.

### 3.3.2 Automatic Generation of Dataflow Diagrams

The third feature of the second-generation tools is automatic production of systems-analysis diagrams. AUTO-DFD and FLEDGED emphasized automatic layout and tool flexibility in the second-generation-tool problems respectively, but they are still diagram-editing tools. They can't automatically generate dataflow diagrams and users have to issue editing command to construct DFDs. *Mondrian* is a system for automatic generation of dataflow diagrams [24].



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The user of the tool defines formally the logical structure and requirements of an information system by using SPSL/SPSA (simple problem statement language/simple problem statement analyzer), store this description in a database. Mondiran accesses the SPSA database to retrieve system flow information and produces an adjacency list which describes the relationship between each object. The placement and routing strategies, encapsulated in module *Produce layout* as shown in Figure 3.3.2–1, is recorded by the adjacency list as it is determined. The graphical information is stored in module *Store data* which can be accessed by both *Extract data* and *Draw DFD* modules.

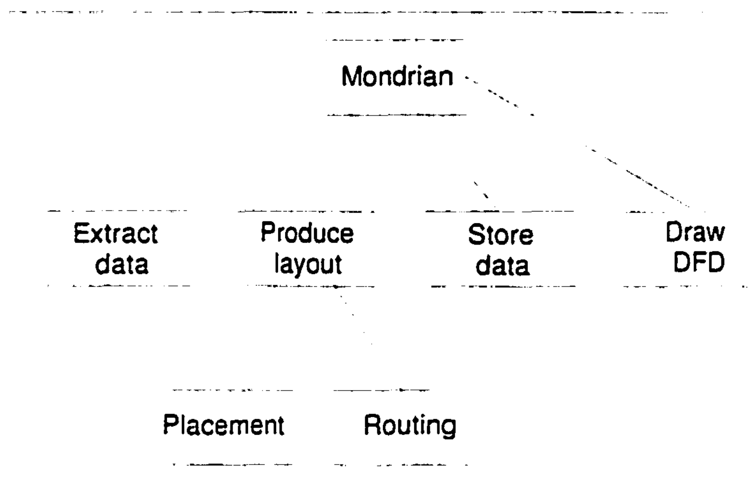



Figure 3.3.2-1 High-level structure of Mondrian

A critical issue in automatic DFD generation is the layout methodology that makes possible the automatic drawing of dataflow diagrams. Batini et al [] in 1986 presented a proposal of a layout algorithm.

The underlined layout algorithm receives as input an abstract graph, specifies connectivity relations between the elements of the diagram, and produces as output



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a corresponding diagram according to the aesthetics. The basic strategy is to build incrementally the layout. First, a good topology is constructed with few crossings between edges. Subsequently, the shape of the diagram is determined in terms of angles appearing along edges. And finally, dimensions are given to the graph, obtaining a grid skeleton for the diagram.

From an aesthetic point of view, an acceptable DFD used in real-life applications has the following properties:

A1: minimization of crossings between connections.

A2: minimization of the global number of bends in connection lines.

A3: minimization of the global length of connections.

A4: minimization of the area of the smallest rectangle covering the diagram.

A5: placement on the external boundary of symbols representing interfaces.

A1 and A5 refer to topology, A2 to shape, A3 and A4 to metric. These fact implies a hierarchic layout representation, where these properties are successively considered. The above aesthetics are generally not compatible. But a priority order can be established to balance these characteristics by using a mathematical model. This model defines three graphs: *plane graph*, *orthogonal graph* and *grid graph*. These graphs are mathematically associated. If two grid graphs have the same grid representation, they have also the same orthogonal representation. If two orthogonal graphs have the same orthogonal representation, they have also the same planar representation. As a consequence, the three representations are hierarchically related, and each representation level is a refinement of the previous one.

The layout algorithm for dataflow diagrams takes as input a DF-graph G =



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(V, E) and produces a planar representation P taking into account aesthetics A1 and A5. Then an orthogonal shape is given to the planar representation finding an orthogonal representation H with the minimum number of bends (aesthetics A2). Finally, a grid representation Q with minimum connections length is embedded into the orthogonal representation according to aesthetics A3 and A4. This last step is also known as compaction.

A CASE tool using the layout algorithm for DFD can syntactically reconstruct a DFD in terms of graph aesthetics. But the preliminary is that there must have existed a dataflow diagram before it is reconstructed. This tool is nothing more than an eye-pleasing improvement of existing dataflow diagram or just a better arrangement of symbols in a dataflow diagram.

### 3.3.3 Executable Dataflow Diagrams

The CASE tools that we have discussed so far are all the DFD-editing tools which can support good-quality documentation, simple forms of consistency checking and bookkeeping either automatically or semi-automatically. It is DFD users responsibility to implement the behavior of the system described by the dataflow diagrams. Converting dataflow specification into executable code is another field studied by system analysts and designers.

Webb and Ward invoked the research interest in executable dataflow diagrams in 1986 [33]. A critical issue in executing dataflow diagrams is to solve concurrency problems. Webb proposed the cycle of distinct time periods as a solution based on Ward's *transformation schema* [32]. The model for execution provides both for functional execution of the logic associated with the individual transformations of the transformation schema and also for the "symbolic" execution of



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

an overall schema [32]. The latter execution is via token-passing similar to the approach described in Petri Net model [13].

Figure 3.3.3–1 shows an alternative model of executable dataflow diagrams which features multiple processing units to achieve concurrency [7]. Each processing unit can handle a single instruction at a time and is fireable when all the operand flows for that instruction are available. The resulting flows become inputs to other instructions or machine outputs. During execution the dataflows are consumed by the instruction and are not then available for use elsewhere, which means that there is no concept of stored variable.

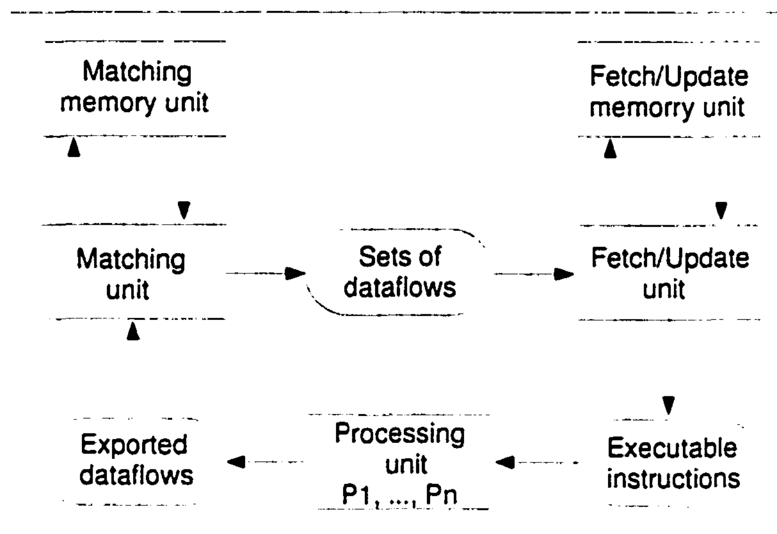



Figure 3.3.3-1 An architecture of dataflow machine

*Matching unit* takes the output dataflows from the processing units and forms them into matching sets, where a set comprises all the dataflows required by an instruction and is represented by a set of process numbers. *Fetch/Update unit* takes each set of dataflows and incorporates it into a copy of the consuming



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instruction, which also contains information on the destinations of the instruction output, to form packet. Each process in the pool of *Processing unit* is able to execute one instruction, or packet, at a time. The method of allocating packets to processors varies from system to system.

A dataflow diagram is a purely functional graphic specification and is an abstract high level design of the system. It is difficult to generate procedural code from the entirely non-procedural diagrams because the DFD itself provides no information about the organization of procedures, the order of their execution, or how the data is to be passed between them (the way the data is passed depends on the implementation). Minor changes in high level specification may require a complete redesign of the corresponding procedural program. Meeson has developed a system that can translate a graphic dataflow diagram into executable code [18].

In Meeson's system, two essential tools for dataflow programming are a graphics editor to create and modify dataflow diagrams, and a compiler to convert diagrams into executable code. The compiler analyzes the connectivity of a dataflow diagram and constructs an abstract syntax tree for function definition (a language used in the system). Unlike Webb's model which uses control information to interpret the procedural behavior of a dataflow diagram, Meeson' model adds a so-called translation "hints" to the dataflow diagrams to solve ambiguities. These hints do not include procedural control information and are not included in printed diagrams either, but are easily accessible through the editor. For example, the hints of the system allow numbering input dataflows in the order they should appear in corresponding function argument lists.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 3.3.4 Executing Dataflow Diagrams

While translating dataflow diagram into executable code can save programmers a lot of implementation time, executing a dataflow diagram can help systems users, analysts and designers to observe the dynamic behavior of the modeled system, understand the system well and consequently modify the design or specifications to fit the requirements well. Based on the development of formal specification of dataflow diagrams, CASE tools for executing dataflow diagrams came out in late 1980s.

Reilly and Brackett' paper on executing dataflow diagrams is among the early jobs done in the area [25]. Their objectives are to determine the requirements for SA support tools that will assist both users and analysts in verifying that a model is semantically correct and consistent, and to investigate feasible design approaches for developing SA support tools meeting the requirements.

The traditional execution of dataflow diagrams was done manually by both system users and analysts with pencils marking the sequence numbers on the processes (transformations) that were activated in response to the external events. The manual execution, frequently called a "playthrough", is often tedious and error-prone for even the smallest model and infeasible for larger models.

In their paper, Reilly and Brackett defines the execution of DFD as tracing the processing that occurs within the system when external events occur. Automatic execution begins with the analyst or user placing a token onto the SA model diagram displayed on the workstation using a mouse or other pointing device. The executable model then "consumes" the token and removes it form the diagram. The "receiver" processes for that token are executed, and they automatically



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

produce new tokens on their output dataflows. This proceeds without further input from the user or analyst, and is animated on the workstation SA diagram.

Reilly and Brackett described a visualization of executed dataflow diagrams and discussed a few models used or possible to be used in execution system, but didn't presented them in details. This job was not done until Fuggetta et al published a paper in 1993 [11]. They introduced an executable visual language (VLP) for formal specifications and prototyping which integrated ER and DFD diagrams in a semantically rigorous and clear way.

To represent synchronization and control conditions explicitly in dataflow diagrams, they proposed a formal dataflow diagram model (FDFD) where data exchanged between functions are represented systematically by boxes, thus eliminating the need for the data sources, sinks and stores of the original DFD model. A data transformer is enabled for activation if and only if all input boxes are full and all blocking output boxes are empty.

The VLP language is based on the FDFD model where it deals with data transformation; it also includes a formal notation for the definition of the types of data contained in the boxes and of the functions associated with bubbles of the diagrams. Being formal, the notation is executable: it is actually a very-high-level language suitable for rapid prototyping.

Data type is defined in a way similar to what is done in Pascal-like languages, starting from elementary types (boolean, integer, and real numbers, characters etc.) and using the usual aggregate constructors, array and record. Functions are defined in a strongly-typed high-level language. Such functions are external to one another, thus the header of a function declaration will contain a list of the function's input and output parameters, according to the following



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

pattern: function *<function_name>* (input *<input_parameter_list>*; output *<output_parameter_list>*);. Function will contain a declarative part where local variables may be defined, and an executable part consisting of composition of the usual instructions of structured programming (assignment, conditional and branching instructions, iterative instructions, function calls). No local function declaration is allowed, and no recursion, either direct or indirect, is admitted.

A graphical user interface is provided in the executing system, which allows user to enter specification in a very easy way. The editor allows the designer to navigate across a refinement tree via "zoom in" and "zoom out" operations that can be applied to different data transformers. Data type definitions are entered through dialogue boxes that guide the designer in the definition process. A text editor can be used to associate narrative comments with the objects of a VLP diagram. The interpreter is activated via a menu option and performs consistency checking and determines the set of terminal data transformers and then starts the execution.

## 3.4 Formal DFD Specifications in CASE Tools

Quite a few CASE DFD support tools were developed to facilitate the use of dataflow diagrams in systems specifications and design. However, the CASE tools have not been used that often as they were expected to. One of important reasons is that the lack of formal framework in dataflow diagrams resulted in CASE tools not powerful enough to handle various needs of DFD users [9]. Most of the tools developed so far are just editing tools. Some tools can generate executable code, but only a small portion of the implementation [14].



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Tse and Pong proposed a formal foundation for DeMarco dataflow diagrams in 1989 [30]. They added a mathematical structure to dataflow diagrams. The model they used is Petri nets [13]. Petri nets can be represented both graphically and algebraically. The graphical representation closely resembles dataflow diagrams and the algebraic representation provides a theoretical basis for the analysis of a specification. Their specification language is called *formal data flow diagrams* (FDFD). Two equivalent forms of FDFD are defined as graphic and symbolic respectively. The graphic representation retains the user-friendly advantages of he original dataflow diagrams and the symbolic one makes use of the algebraic foundation of Petri nets. FDFD also has a formal syntax so that it can be processed easily by a computer.

FDFD defines a 4–tuple G = (D, T, I, O) where

D is the set of dataflows,

$T = \{t_1, t_2, \ldots, t_n\}$, where $n \geq 1$, is a finite set of tasks,

D and T are disjoint.

$I: T \rightarrow E$ and $O: T \rightarrow E$ are functions which map tasks to dataflow expression, I is called the input logic function and O the output logic function.

The notations of token and firing from Petri nets are also incorporated in FDFD to model the behavior of a systems dynamically. The presence of a token means that input through a given dataflow is ready for task. Marking of a FDFD is a function $u: D \rightarrow N$ from the set of dataflows D of a FDFD to the set of non-negative integers N. Given a FDFD G and a marking u, the ordered couple M = (G, u) is called a marked FDFD. A marking v is said to be *reachable* from another marking u if there exists a sequence of executions that changes u into v.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

These dynamic elements provide the basis for analyzing the dynamic behavior of the system. The analysis will help to detect problems which may not otherwise be apparent in the static model, such as deadlocks or tasks that will never be activated. Three types of consistency analysis can be achieved through FDFD — global consistency, structural consistency and behavioral consistency.

Another effort for developing formal specification of dataflow diagrams was made by France in 1992 [8]. He described a method for associating a DFD with a formal specification. The intention is to enhance the use of the DFD as a formal specification tool that can be used to document application functionality in a understandable manner. Meanwhile, this tool should be capable of producing a formal specification that can be used to evaluate semantic properties of the application.

The formal specification used by France is based on the algebraic specification technique. A semantically extended DFD (ExtDFD) is defined as a control-extended DFD (C-DFD) [32] associated with formal semantics. ExtDFD thus have two aspects: syntactic and semantic. The syntactic aspect of an ExtDFD is its graphic representation and the semantic aspect is a behavioral interpretation of its C-DFD. The basic interpretation of C-DFD is classified as data domain, data flows, data stores and data transforms.

In the model, a dataflow is interpreted as, either an asynchronous or synchronous data interface between its generator and receivers. A synchronous dataflow requires its generator and receivers to cooperate for the data sending and receiving, but asynchronous one doesn't. A set of well-formed statements are defined as axioms to interpret the state transition semantics. The dynamic behavior of ExtDFD is described by activation and deactivation of data transforms.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 4 SEMANTIC DESCRIPTION OF DFDS

From what was described in last section, we can see that CASE tools for DFDs are still in a preliminary stage with concentration on editing and graphical representation. All these tools lack formal specifications though a few tried to give DFDs a formal foundation. The basic requirement for a formal representation of DFDs is that the underlined language must be capable of describing concurrency and functional model. A good candidate is so called Calculus of Communicating System (CCS) [31] which draw wide attention in software engineering from academic institution and from industry to some degree.

## 4.1 Calculus of Communicating System

CCS is a language that may describe the various ways in which cooperating sequential processes can interact with each other. The examples of typical precesses are: receive, send, and retransmit processes in the X.25 link- level: arbiters and mutual exclusion elements in asynchronous hardware design; boats, trucks, cranes in a descrete-event harbor simulation; etc. Such simulation processes can map directly into CCS processes, one for one [20].

Communication and concurrency are complementary notion, both essential in understanding complex dynamic systems. On the one hand such a system has diversity, being composed of several parts each acting concurrently with, and independently of, other parts; on the other hand a complex system has unity achieved through communication among its parts.

Underlying both these notions is the assumption that each of the several parts of such a system has its own identity that persists through time. These parts are



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

termed as agents that are the basic objects in CCS. They may be constructed by prefixing '.', non-deterministic choice '+', parallel composition '|' and restriction '\{}'.

### 4.1.1 Sequencing

A simple agent has an inflow and an outflow associated with its two ports as shown in Figure 4.1.1-1. CCS representation can be constructed as:

$C = inflow.C'$ and $C' = \overline{outflow}.C$

The notation '.' stands for sequential ordering of actions. The above notation can be rewritten in a recursive way like:

$C = inflow.\overline{outflow}.C$

By convention, output actions are given co-names in the way that two communicating agents have consistent relation. For example, a system described in Figure 4.1.1-2 can be represented as:

$P = a.\bar{b}.P$ and $Q = b.\bar{c}.Q$

Where $b$ and $\bar{b}$ are exactly the same action. When the action $\bar{b}$ is fired by P, Q takes in the same action $b$ at the same time.

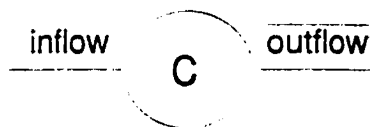



Figure 4.1.1-1



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

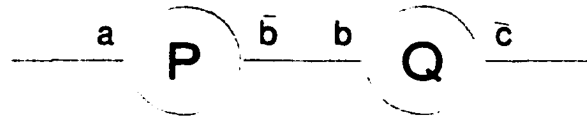



Figure 4.1.1-2

### 4.1.2 Choice

Choice notated as '+' is used to represent non-determinictic alternatives. Figure 4.1.2-1 shows an example. Agent C has two alternative inflows 'a' and 'b'. One choice of the action sequence in CCS code is: $R = a.\bar{c}.R$. The other one is: $R = b.\bar{c}.R$. Which action course agent R should take depends on the competition between inflow 'a' and 'b' when agent R is ready for receiving input. This kind of event can be represented in CCS by using notation '+' as:

$$R \;=\; a.\bar{c}.R \;+\; b.\bar{c}.R$$

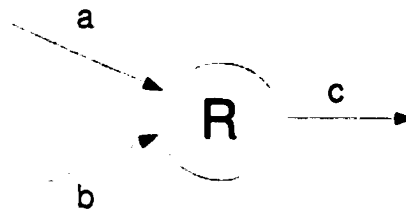



Figure 4.1.2-1

### 4.1.3 Parallel Composition

We use Figure 4.1.1-2 to describe the complementary actions 'receive' and 'send'. Action '$\bar{b}$' is the complementary of action '$b$'. Now comes the question how agent P or Q interacts with each other. CCS uses another constructor '|', called composition, to express the interaction between agents. Two agents, which interact with each other, can be composed into one agent by using this



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

function. Thus if P and Q are agents then P |Q is an agent which represents the parallel composition of P and Q in such a way that each of P and Q may proceed independently of the other but may also communicate through the complementary actions '$\overline{b}$' and '$b$'.

A transition of the form E -> E' indicates that agent E can perform the action x and becomes E'. consider the composition $(x.E) \mid (\overline{x}.F')$. If the agent $(x.E)$ performs the action x and becomes E and, simultaneously, the agent $(\overline{x}.F')$ performs $\overline{x}$ and becomes F', the composition will become E |F'. This kind of event is expressed by the $\tau$—transition $(x.E) \mid (\overline{x}.F') \xrightarrow{\tau} E \mid F'$. By using parallel composition, Figure 4.1.1-2 can be represented as: $E = (P \mid Q) \setminus \{b\}$ where \{b} stands for restriction which means that agent P and Q interact with each other through action b.

## 4.2 Semantic Representation of DFDs

Calculus of Communicating Systems (CCS) is selected to represent the semantics of a DFD. Each node of a DFD is associated with an agent and each arrow in a DFD is associated with communication between agents. Four of CCS functions are used to construct the logical structure of a DFD. We use Figure 4.2–1 to describe how the semantics of DFDs can be expressed.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

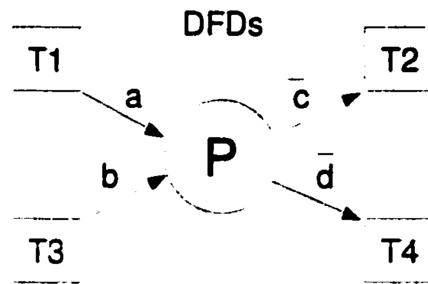



Figure 4.2-1

1. **Sequence** operator represented by " . " is interpreted as actions taking order. One of the action sequence for Process P shown in Figure 4.2–1 has the semantics: $P = a.\bar{d}.P$ where action of "receiving d" represented by "$\bar{d}$ " is the complementary action of "sending d" represented by "d". This complementary notation is for synchronization purpose. This CCS code means that Process P receives inflow a, process it and then sends outflow d.

2. **Or** operator represented by " + " is interpreted as options. Process P has the semantics: $P = a.(\bar{c} + \bar{d}).P + b.(\bar{c} + \bar{d})P$. Process P has two inflow choices "a" or "b" and two outflow choices "c" and "d". If inflow "a" succeeds in competition against inflow "b", the action sequence becomes either $P = a.\bar{c}.P$ or $P = a.\bar{d}.P$

3. **Composition** operator represented by " | " is interpreted as system interface. The DFDs shown in Figure 4.2–1 has the semantics: DFDs = (T1 | T2 | T3 | T4)\{a, b, c, d}. This means that DFDs has a interface consisting of T1, T1, T3 and T4 among which the internal actions are hiding from outside of the system.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4. **Restriction** operator represented by " \ "" | " is interpreted as system internal information hiding. In the above example ( . . . )\{a, b, c, d}, data flows "a", "b", "c" and "d" are the system's internal flow which cannot be observed from outside of the system.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 5 A SEMANTIC-DRIVEN DFD PROCESSING SYSTEM

Since DFD is widely used in both software development and reverse engineering, it is of research interest to develop a system which can understand a dataflow diagram. Furthermore, a system which can interpret DFDs will dig up a route to reuse software documents in high-level abstraction.

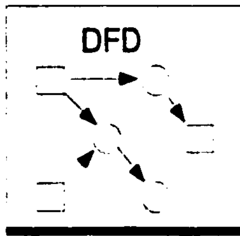
Understanding a diagram requires a number of steps. The system involves two phases: *Recognition* and *Understanding* [2]. The main functions in the recognition phase are scanning the printed document and generate a layout structure of DFDs. Techniques for the recognition phase are fairly well understood although this is still an active area of research. My research interest is concentrated on understanding phase. Figure 5–1 shows a scheme for such a understanding system

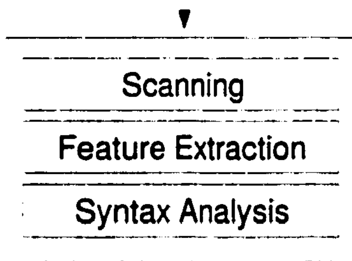
The understanding phase consists of two independent subsystems. One of them takes layout structure of DFDs as input and generate corresponding logical structure. The other one then takes logical structure as input and accomplishes a couple of tasks which include: simulation and equivalence checking. Simulation can simulate the underlying systems behavior by executing a graphic DFD. Equivalence checking can compare two data flow diagrams to determine whether they are semantically equivalent or not.

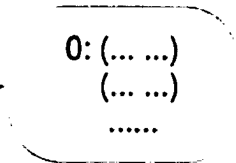


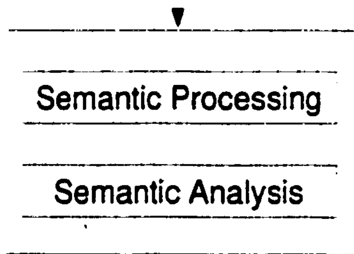
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

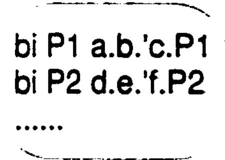
**System for Understanding DFDs**



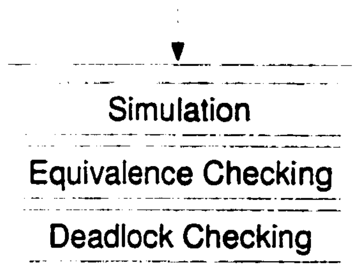




Figure 5-1



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 6 SYSTEM SPECIFICATION

## 6.1 General Description

The underlined system to be developed is a dataflow diagram processor which can semantically understand a DFD and provide the user with some useful services. The system is called *DFDPRO*. The purposes of developing a system like *DFDPRO* are to assist system analysts in understanding the behavior of a system and its subsystem, to assist system maintainer in adapting existing system to fit new platform, to assist system developers in designing brand new systems.

### 6.1.1 The Purpose of DFDPRO

DFDPRO is a semantics-driven dataflow diagram processor that allows the user to observe the behavior of a dataflow diagram through a simulation process and analyses a dataflow diagram through comparison, deadlock detection and state space checking. It can process DFDs which has hierarchical structure. It can decompose a DFD into several sub DFDs. Its resource requirements are kept at minimum and the commands are kept as simple as possible. DFDPRO provides the user with a graphic based simple but full-featured interface and is developed with some goals in mind in two areas:

***Fast Operation***: DFDPRO is designed to operate quickly, especially in these operations:

- Initial loading a file for translation and simulation.
- Moving through the window.
- Quick access each function.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

***Easy Use***: DFDPRO is simple and user friendly

- All functions are displayed on the top of the screen.
- All sub-functions are organized in pull-down menu.
- Every function is easily understood .
- On-line help is provided.

### 6.1.2 Product Functions

DFDPRO offers the following functions:

***File Operations:*** DFDPRO allows the user to *open* an existing document that is on the disk, *save* the current document that is in the main window, *create* a new document, *cut* a file that is on the disk, *print* document and *exit* the system.

***Editing Operations:*** Editing functions allow the user to *delete* a portion of an opened file, *copy* and *paste* or *cut* and *paste* the contents of the current file, and *undo* the previous operations.

***Translating Operations:*** Translating operations allow the user to choose the data file from a file list and convert the file into a CWB code file. CWB stands for Edinburgh Concurrency Workbench. It is an automated tool which caters for manipulation and analysis of concurrent systems. CWB grammar is based on CCS (Calculus of Communicating System) which is used to describe the semantics of DFDs. The detailed description of CCS can be found in Appendix D and CWB in Appendix E.

***Simulating Operations:*** Simulation operations allow the user to do *simulation* on a dataflow diagram, to check observational *equivalence* between two data flow diagrams based on the underlying formal semantics, to detect whether a



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

*deadlock* will occur or not and to find the *minimal state space* of a dataflow diagram.

***Help Function:*** Help function gives on screen a brief explanation of all functions and shows the user how to use these functions.

## 6.2 System Model

### 6.2.1 The Logical Structure of the System

The system consists of two independent subsystems which communicate with users through a common graphical user interface (GUI). One of them is a DFD translator which takes layout structure of DFDs as input and generate corresponding logical structure. The other one is a simulator which then takes logical structure as input and accomplishes a few tasks such as simulation, equivalence checking and deadlock detection. The logical structure of the system is shown in Figure 7.2–1 [3].



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

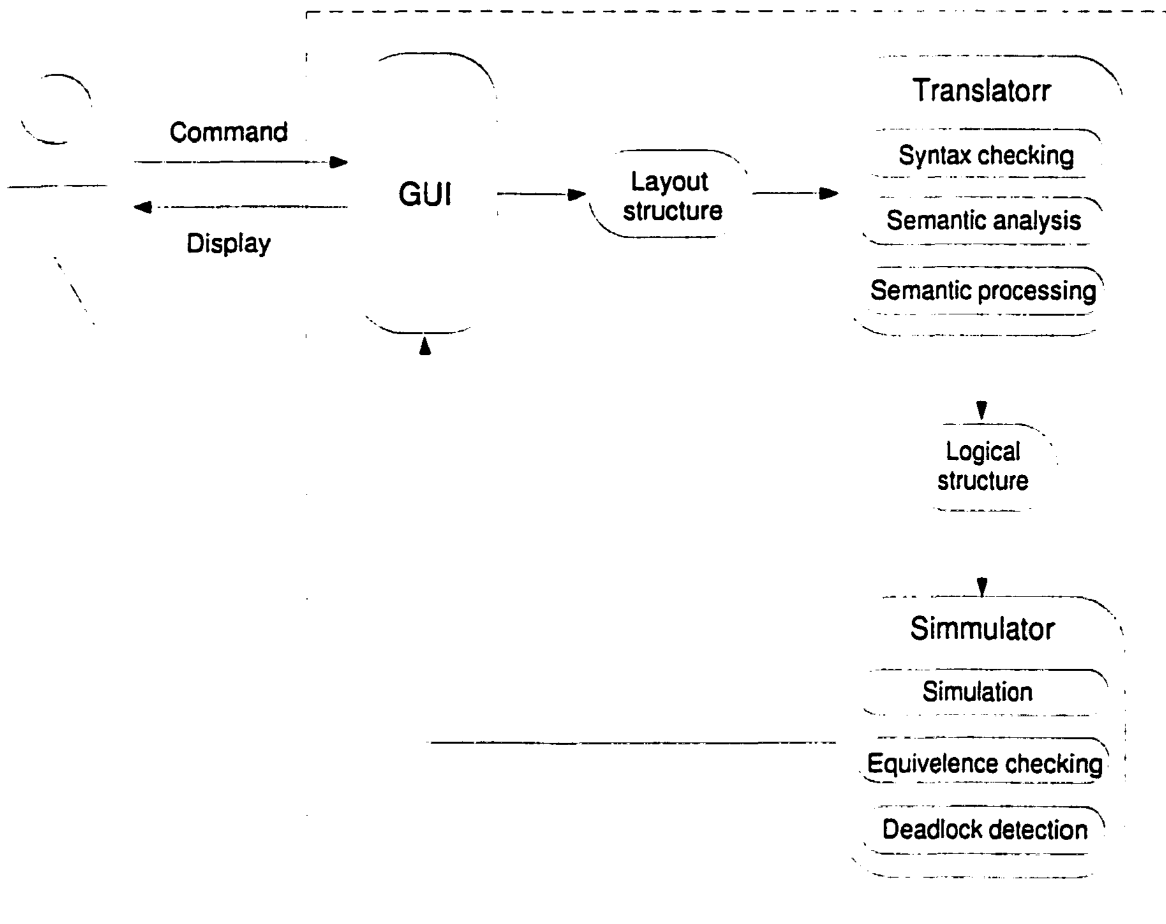



Figure 6.2.1-1: The Logical Structure of the System

The Compiler consists of two major components: a recursive descend *parser* which checks both syntactic and structural correctness of the tuple representations in layout structure and a *translator* which converts a DFD in layout structure format into the logical structure format in CWB code.

## 6.2.2 Display Description

The main window of DFDPRO is shown in Figure 7.3–1. The menu bar



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is at the top of the window. It displays all the functions that the user can use. Each button on the menu bar handles a set of operations in the corresponding task domain and has a hierarchical menu structure. DFDPRO provides scroll bar (left, right, up and down) when the **document** or the **graph** in screen is larger than the window area. The main window can display either a graph if the command issued by the user is Simulating, Equivalence, Deadlock and Minimal Space or a document if the command is not these in Simulation submenu.

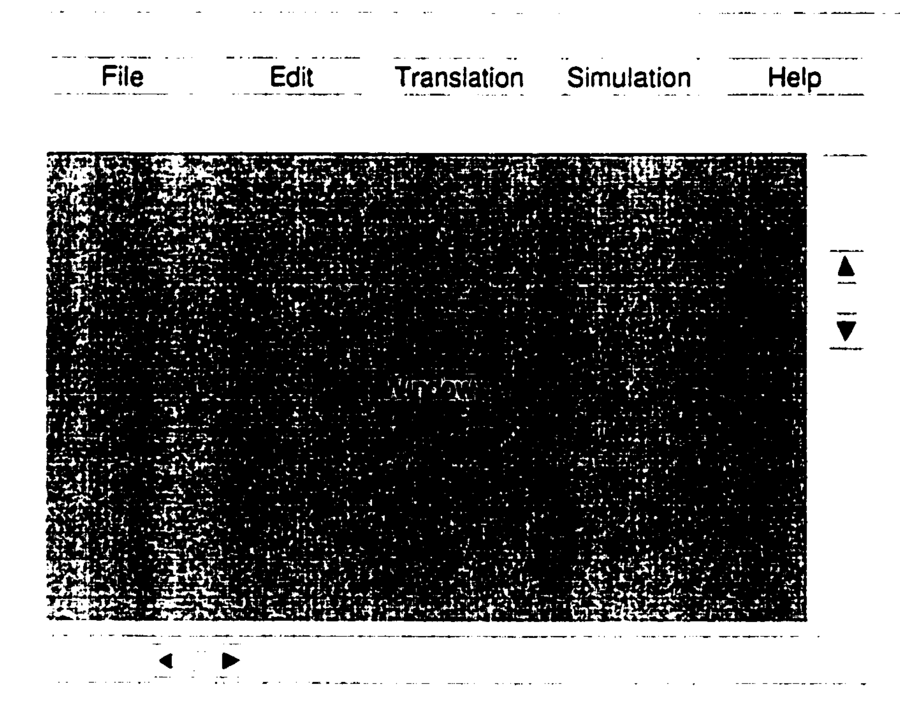



Figure 6.2.2-1: Graphical User Interface

## 6.2.3 The Workspace

The major work space is the graphic user interface — the main window shown in Figure 6.2.2–1. DFDPRO allows at most two separate workspaces. The second



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

workspace other than main window has the format **subwindow**. But DFDPRO allows multiple list-box windows, dialog windows or message windows. Each window can be moved around screen and resized.

## 6.3 System Services

The design of file handling service and editing service is trivial. My main concern is to develop a grammar for DFD syntax checking and a language to describe the semantics of DFD. The grammar I developed is a LL(1) grammar. It represents DFDs with hierarchical structure. The language I used to describe the semantics of DFDs is based on Calculus of Communicating Systems (CCS). There are two reasons to choose CCS. First of all, CCS is a formal semantic description for a concurrent system which is a super set of DFDs model. Secondly, there is a tool called Edinburgh Concurrency Workbench available in schoenfinkel, which is based on CCS and can do a lot analysis of a concurrent system.

### 6.3.1 Functional Requirements

The basic functions DFDPRO performs are organized in a hierarchical structure to make the user easier to access each function. Figure 6.3.1–1 shows the logical structure of the system functions.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

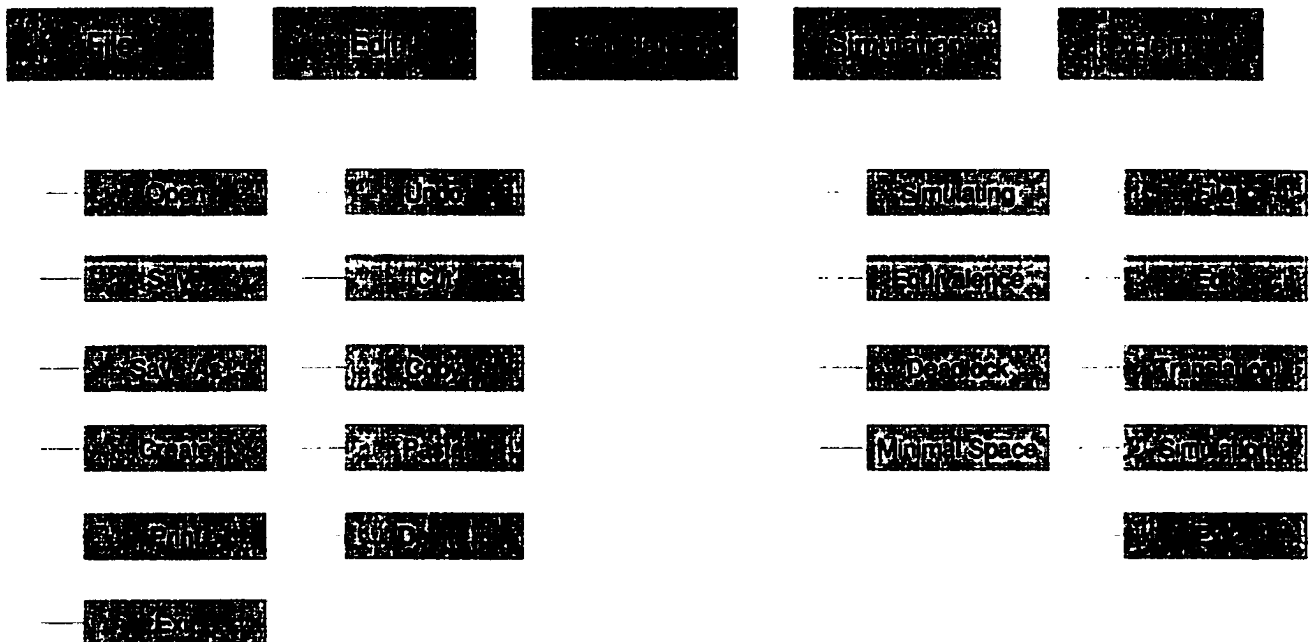



Figure 6.3.1-1: The Logical Structure of the System Functions

## 6.3.2 Translation Services

**Translation Command** Translation command invokes the translator which takes DFD tuple representation as input and translates it into CWB code as output.

**The Input of Translator** The DFD tuple representation in layout structure is the input of the translator. It represents the syntax of a DFD and must satisfy the following requirements.

**Basic Assumptions**

- The character sequence of input must syntactically satisfy the LL(1) grammar.
- Every entity and every flow must have a unique identifier.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- The input should be read from a file with extension *.dfd* on disk.

**LL(1) Grammar**

a. DFDs :: = Identifier (Diagram) DFDs'

b. DFDs' :: = , Identifier (Diagram) DFDs' | $\epsilon$

c. Diagram :: = (Tuple) Tuple'

d. Tuple' :: = , (Tuple) Tuple' | $\epsilon$

e. Tuple :: = Type Identifier Relation

f. Relation :: = (List) Relation' | Relation' | $\epsilon$

g. Relation' :: = EXTERNAL_I (Flow) | EXTERNAL_O (Flow) | $\epsilon$

h. List :: = (Identifier, Flow) List'

i. List' :: = , (Identifier, Flow) List' | $\epsilon$

j. Type :: = SOURCE_TERMINATOR

k. | SINK_TERMINATOR

l. | PROCESS

m. | DATA_STORE

n. | AUXILIARY_SPLIT

o. | AUXILIARY_MERGE

p. Identifier :: = (A | B | . . . | Z) (A | B | . . . | Z | a | b | . . . | z | 1 | 2 | ... | 9)*

q. Flow :: = (a | b | ... | z)+ Flow'

r. Flow' :: = , (a | b | ... | z)+ Flow' | $\epsilon$

**DFD Layout Structure** consists of three levels: diagram tuple, node tuple and successor tuple. The DFD name, the name of node and the name of successor must start with upper-case letter. There must and only have space



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

between diagram tuples. Node tuples start with "(Type" and must be separated by comma. Successor tuples start with "(Identifier" or "EXTERNAL_I" or "EXTERNAL_O" and must be separated by comma.

- Representing a diagram of a DFD and containing information about the diagram, each diagram tuple has the following structure:

    DFD Name ((node1), (node2), . . . )

- Representing a node of the diagram and containing information about a node, each node tuple has the following structure. If there are external flows to or from the node, the format "(successor)" could become "EXTERNAL_I ( flow, flow, . . . )" or "EXTERNAL_O ( flow, flow, . . . )".

    Type Name of node ((successor1), (successor2), . . . ))

- Representing a successor of the node and containing information about the successor, each successor tuple has the following structure:

    Name of successor, the label of the data flowing into it

**An Example of the Layout Structure of DFDs:** For reader to well understand the content of the document, I raise an example of hierarchical DFDs as a standard model to describe the services and functions DFDPRO provides. The DFDs shown in Figure 6.3.2–1 has three levels and four diagrams. By



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

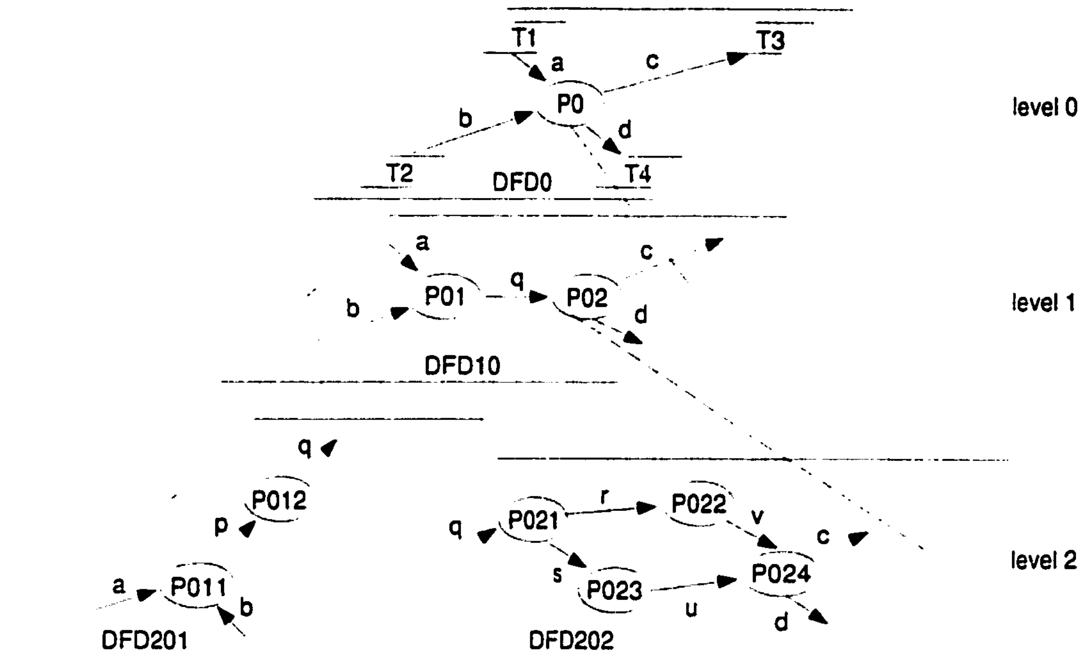



Figure 6.3.2-1 An example of hierarchical DFDs

using the above LL(1) grammar, the tuple representation of levelled DFDs shown in Figure 6.3.2–1 can be given bellow:

a. **Level 0**

DFD0((SOURCE_TERMINATOR T1 ((P0, a))),
(SOURCE_TERMINATOR T2 ((P0, b))),
(PROCESS P0 ((T3, c), (T4, d))),
(SINK_TERMINATOR T3),
(SINK_TERMINATOR T4))

b. **Level 1**

DFD10((PROCESS P01 ((P02, q)) EXTERNAL_I (a, b)),



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(PROCESS P02 EXTERNAL_O (c, d)))

c. **Level 2**

DFD201((PROCESS P011 ((P012, p)), EXTERNAL_I (a, b)),
(PROCESS P012 EXTERNAL_O (q))
DFD202((PROCESS P021 ((P022, r), (P023, s)), EXTERNAL_I (q)),
(PROCESS P022 ((P024, v))),
(PROCESS P023 ((P024, u))),
(PROCESS P024 EXTERNAL_O (c, d)))

### The Output of Translator

The CWB code of DFD in logical structure is the output of the translator. It represents the semantics of a DFD and must satisfy the following requirements.

#### Basic Requirements

- The character sequence of output must satisfy the CWB syntax.
- Every entity and every flow must have a unique identifier.
- The output should be written to a file with the same file name as the input file but different extension which is *.cwb*.

#### DFD Logical Structure

- Representing the semantics of the nodes, the composition node in the logical structure has the following format:

(Agent1 | Agent2 . . . ) \ Restrictions



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Among the format, agent has the equation defined recursively as agent expression which consists of sequence of actions the agent takes or as option of agent expressions:

  Agent = a.'b. . . . Agent or

  Agent = a.'b. . . . Agent + c.d . . . Agent' + . . .

- The Restrictions consists of sequence of internal actions between Agent1, Agent2 . . . and has the format { f, g, h, . . . }.

**An Example of the Logical Structure of DFDs** shown in Figure 6.3.2–1 is given bellow. It is expressed in CWB code based on CCS notations. The complementary actions are represented as " 'action " instead of $\overline{action}$.

a. **Level 0**

```
DFD0 = (T1 | T2 | T3 | T4)\{a, b, c, d}
   T1 = input.'a.T1
   T2 = input.'b.T2
   T3 = c.'output.T3
   T4 = d.'output.T4
   P0 = a.('c + 'd).P0 + b.(,c + 'd).P0
```

b. **Level 1**

```
P0  = (P01 | P02)\{q}
P01 = a.'q.P01 + b.'q.P01
P02 = q.'c.P02 + q.'d.P02
```



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

c. **Level 2**

P01 = (P011 | P012)\{p}

P011 = a.'p.P011 + b.'p.P011

P012 = p.'q.P012

P02 = (P021 | P022 | P023 | P024)\{r, s, u, v}

P021 = q.'r.P021 + q.'s.P021

P022 = r.'v.P022

P023 = s.'u.P023

P024 = u.('c + 'd).P024 + v.('c. + 'd).P024

**Translator**

The translator is a component of the system responsible for

1. Reading file with extension *.dfd* from disk,
2. Parsing the tuple representation of DFD in layout structure,
3. Building a parsing tree for each dataflow diagram with each node containing information about the node,
4. Checking the syntax of the tuple representation in terms of the LL(1) grammar,
5. Giving error message if syntax error is detected and terminating the translation process,
6. Converting the tuple representations into CWB code in DFD logical structure in terms of translation rules,
7. Writing the output file with extension *.cwb* on disk.

**Translation Rules**



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DFD logical structure is generated by traversing the DFD parsing tree. The translator generates CWB code for each node traversed. The parsing tree is traversed level by level. For each node being traversed, the translator generates code based on the type of the node.

1. If the node is a process, actions corresponding to receiving and sending data must initially be captured. The convention adopted is that one or more inputs of process suffice to compute the outputs. If a process requires all of its in-flows to compute the outputs, an auxiliary node should be used. The agent expression representing a process is defined recursively and uses the *or* and *sequence* functions. Node (PROCESS P01 (( P02, q)), EXTERNAL_I (a, b)) in DFD10 presented in the example of layout structure is therefore translated into P01 = a.b.'q.P01 presented in the example of logical structure.
2. If the node is a data store, actions representing inputs can be performed independently of the actions representing outputs, since a data store does not perform calculations to derive outputs from inputs.
3. If the node is a source terminator, the only task of this node is to send data to other processes. For synchronization purpose, a special action *input* is introduced. It precedes the action of source's sending the data from to other processes. Node (SOURCE_TERMINATOR T1 ((P0, a))) in DFD0 presented in the example of layout structure is therefore translated into T1 = input.'a.T1 presented in the example of logical structure.
4. If the node is a sink terminator, the only task of this node is to receive data from other processes. For synchronization purpose, a special action *output* is introduced. It takes place after the action of sink's receiving the data from other processes. Node (SINK_TERMINATOR T3) in DFD0 presented in



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the example of layout structure is therefore translated into T3 = c.'output.T3 presented in the example of logical structure.

5. If the node is an auxiliary node, what must be taken into account is all the possible permutations of the input sequences to the auxiliary merge and the output sequences from the auxiliary split. Figure 8.2.6–1 shows an example of auxiliary nodes.

   1. AS is a split auxiliary node which, if i is not an external flow nor a flow from a source terminator, could be expressed in DFD tuple representation as:

      (AUXILIARY_SPLIT AS ((P1, a)), ((P2, b)))

      and can be translated into logical structure as:

      AS = i.'a.'b.AS + i.'b.'a.AS

   2. AM is a merge auxiliary node which, if j and k are not external flows nor flows from source terminators, could be expressed in DFD tuple representation as:

      (AUXILIARY_MERGE AM ((P3, c)))

      and can be translated into logical structure as:

      AM = j.k.'c.AM ÷ k.j.'c.AM

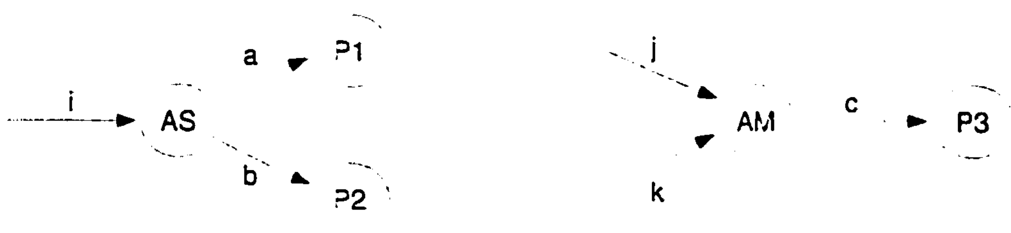



Figure 6.3.2-2: Auxilrary Nodes



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6. The communications between the agents in a specific level of DFDs are represented by agent composition and data flow restriction. Node

   DFD0((SOURCE_TERMINATOR T1 ((P0, a))),
   (SOURCE_TERMINATOR T2 ((P0, b))),
   (PROCESS P0 ((T3, c), (T4, d))),
   (SINK_TERMINATOR T3),
   (SINK_TERMINATOR T4))

   in DFD0 presented in the example of layout structure is therefore translated into DFD0 = (T1 | T2 | T3 | T4)\{a, b, c, d} presented in the example of logical structure.

7. If the node has a refined sub DFD, the data flowing into the node is interpreted as special source terminator — external input in the sub DFD, and the data flowing out of the node is interpreted as special sink terminator — external output in the sub DFD. But both external flows in the sub DFD keep the same identifiers as they have in the higher level DFD.

**Concurrency Workbench**

The Concurrency Workbench (CWB) is a tool that supports the automatic verification of finite-state processes. In particular, CWB allows for various equivalence, preorder and model checking using a variety of different precess semantics. For instance, the processes to be analyzed by CWB can be expressed in CCS notations. The CCS notation used as the input of CWB machine has a little modification in the way that the complementary action is expressed as " 'action " instead of overbar expression "$\overline{action}$ in CCS. Such a variety has no significant meaning more than convenience.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since its powerful features in analysis of concurrent systems, CWB can be used to manipulate and analyze DFDs. Through CWB tool, we can use the formal description of DFDs to reason about the equivalence of two DFDs with quite different layout structure and to simulate the behavior of DFDs. As a matter of fact, CWB is a ported component of DFDPRO. We will identify CWB component in system design section.

### 6.3.3 Simulation Services

Simulation command invokes the simulator which takes CWB code in DFD logical structure as input and then simulates the behavior of the DFD, checks whether two DFDs are observational equivalent, detects whether deadlock can occur and where it occurs, and figures out the minimal state space of the DFD.

#### Simulation Command

This command will load in a *.cwb* file from disk and show a graphic dataflow diagram in the main window. Then the user can interactively perform simulation operation by using mouse.

**State of Dataflow Diagram** is represented by the states of its components. Each component has three states: not active, ready, active which are represented by red color, yellow color and blue color respectively. The user cannot click a component which has red or blue color. If it is clicked, a **Beep** will sound and an **error message** will appear.

a. **Source Terminator**

- Not active means that it has not got its input yet and can not send out data.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Ready means that it got its input and ready to send out data
- Active means that it is sending out data.

b. **Sink Terminator**

- Not active means that it can not receive data.
- Ready means that it is ready to receive data
- Active means that it is receiving data and issuing output.

c. **Process**

- Not active means that it can not process data.
- Ready means that it is receiving data and ready to process it
- Active means that it is processing data and sending out data.

d. **Data Store**

- Not active means that it is closed.
- Ready means that it is open.
- Active means that the process connecting it is sending it data or retrieving data from it.

e. **Data Flow**

- Not active means that there is no data flow.
- Ready means that data flow is available at ports.
- Active means that data flow is going through.

**Transition Between States** for the same component follows the repeated sequence: read →active → not active → ready. Transition between states for different components satisfies the following rules:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Initial States**

- All source terminators are ready.
- All other components are not active.

**States Between Adjacent Components**

- Any two adjacent components must have different states if they all were activated.
- Along the direction a data flow arrow points, the state sequence of any three adjacent components must follow: active → ready → not active → active, if they all were activated.

**States of Auxiliary Nodes**

- For auxiliary merge, the merged flow is not ready until all the in flows are active.
- For auxiliary split, all the split flows are ready simultaneously right after the in flow is active.

**States of Components with More Than One Flows**

- For a component with more than one in flows, it is ready if any of the in flows is active.
- For a component with more than one out flows, any of these flows is ready if it is active

**Refinement of Dataflow Diagram** is done by double clicking a process if its decomposed dataflow diagram is available. If the user double clicks such a process, appears another window with the decomposed diagram which shows initial states.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**An Example of States Transition** is shown in Figure 6.3.3–1 where R (Red) represents state *not active*, Y (Yellow) represents state *ready*, B (Blue) represents state *active*, and a, b represent data flows. Reader can verify the above rules by following the sequence of transitions horizontally along the data flow arrow and vertically along the different states for the same component.

**An Example of Auxiliary States** is shown in Figure 6.3.3–2 where i, j, k represent in-flows and a, b, c represent out-flows.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

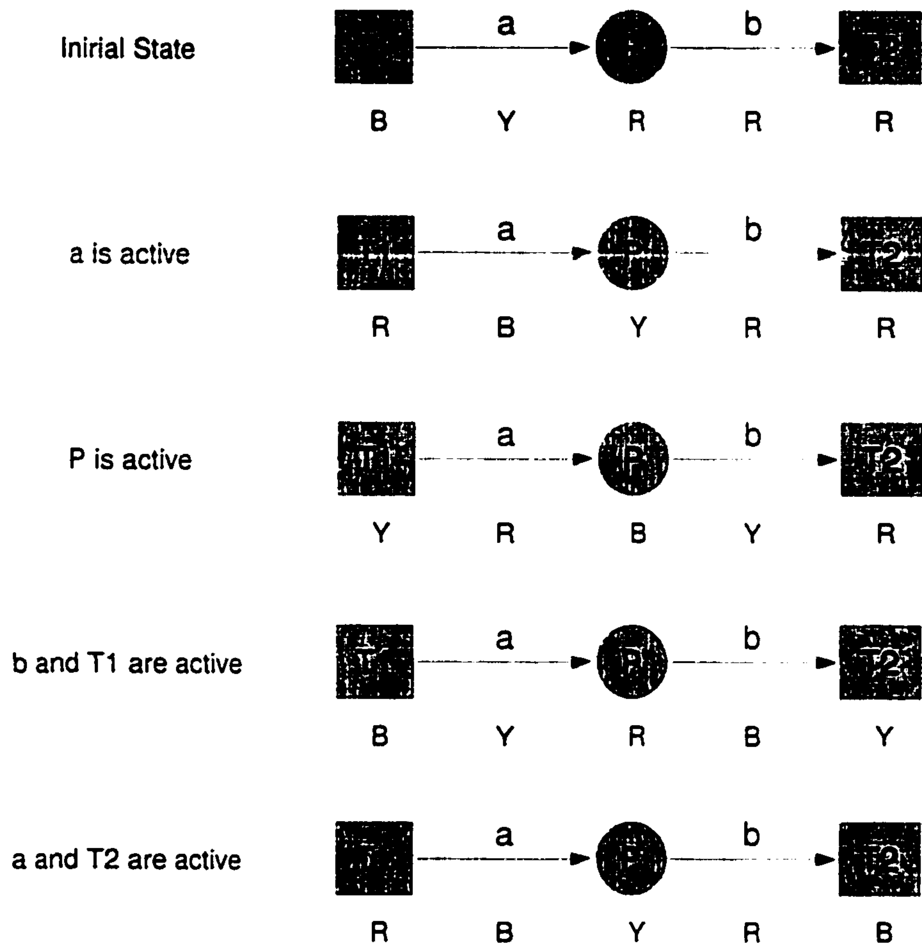



Figure 6.3.3-1: An Example of States Transition



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

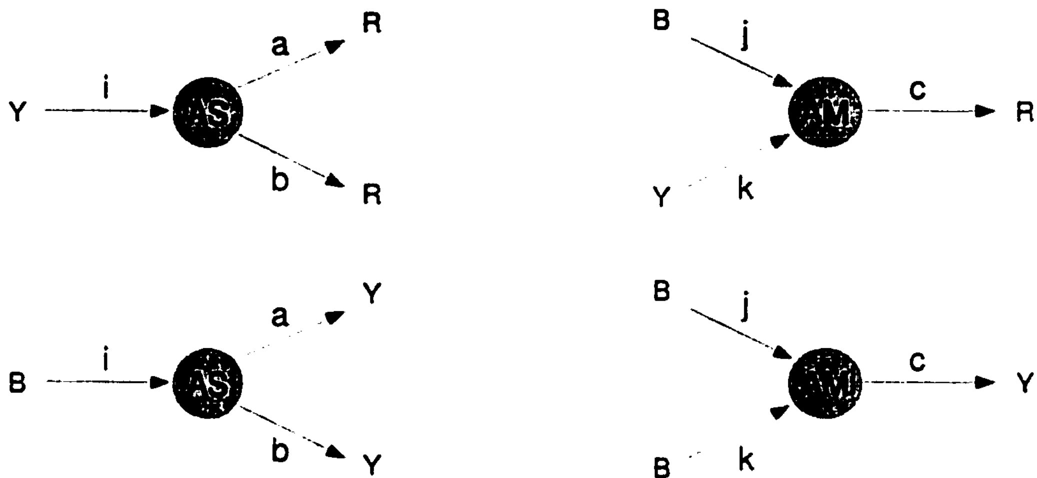



Figure 6.3.3-2: An Example of Auxiliary States

**Equivalence Command**

This command checks two DFDs represented by *.cwb* files on the disk and compares them in terms of the definition of strong bisimulation. If the two compared DFDs are not observational equivalent, the difference will be displayed on screen. Strong Bisimulation can be found in Appendix D.

**Deadlock Command**

DFDPRO can detect deadlock part of a dataflow diagram using this command. If there is a deadlock, the sequence of actions that cause the deadlock will be displayed on screen. The deadlock model used in this command is described as follows.

**Deadlock Definition:** *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.*



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Conditions for Deadlock**

- **Mutual Exclusion:** Each resource is either currently assigned to exactly one precess or is available.
- **Hold and Wait:** Process currently holding resources granted earlier can request new resources.
- **No Preemption:** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released be the process holding them.
- **Circular Wait:** There must be a circular chain of two or more process, each of which is waiting for a resource held by other member in the chain.

**Minimal State Space Command**

This command is used to find the minimum number of the state space of a given dataflow diagram. If this command is executed, the system will generate a new agent representing another dataflow diagram that possesses the smallest state space but is observational equivalent to the original dataflow diagram.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 7 SYSTEM DESIGN

## 7.1 High-Level Description

Figure 6.2.1–1 in section System Specification illustrates the logical structure of the defined DFDPRO processor. It takes quite a few steps for the system to get DFD information from diagrams drawn in papers and accomplish the services that it is supposed to provide. A very high-level flow chart of the system is designed as what is shown in Figure 7.1–1.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

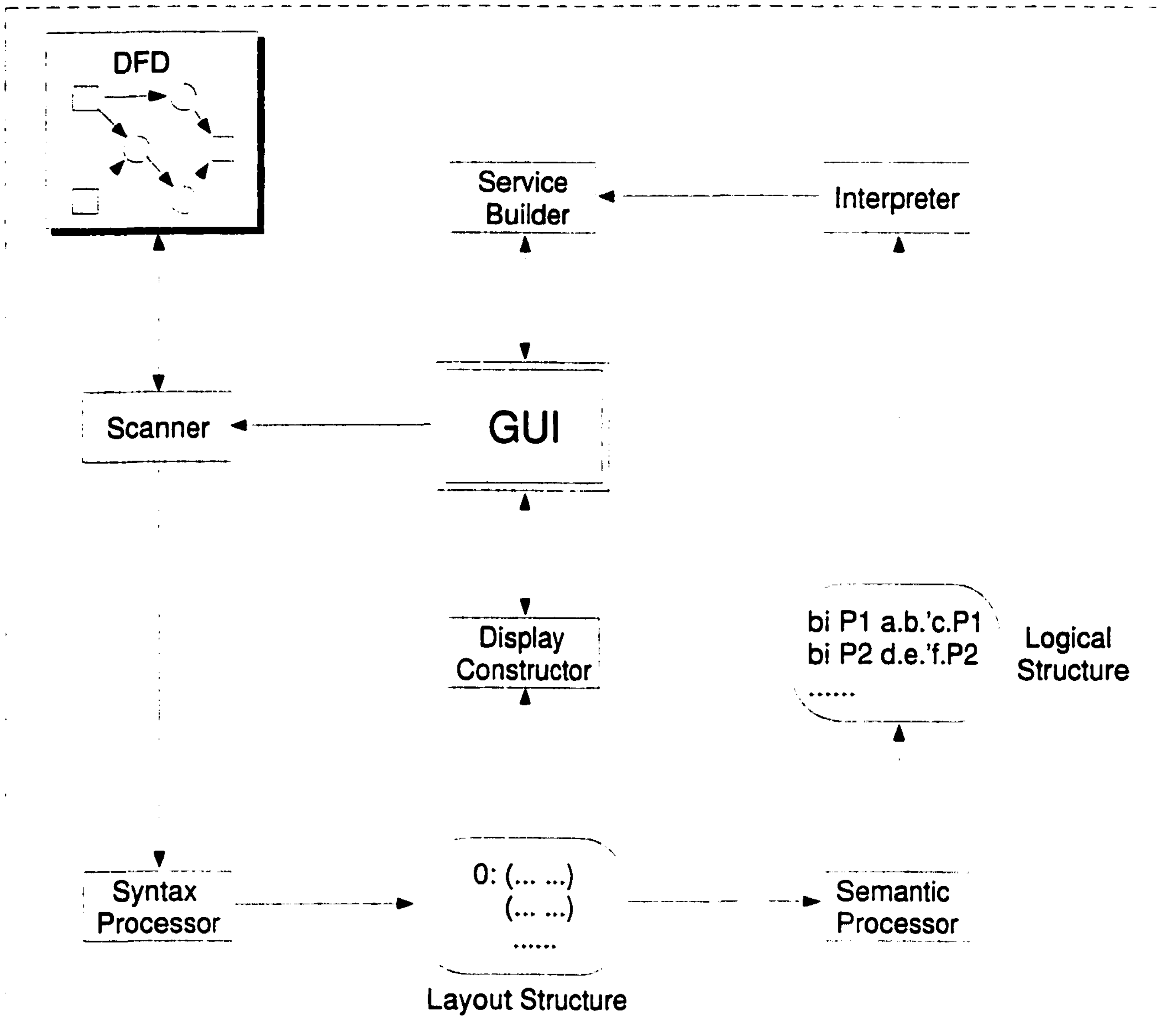



Figure 7.1-1

The graphic user interface scans in a DFD, displays the restructured diagram and provides a set of DFD analysis services. Scanner looks at printed document



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and yields a digital representation of the DFD that is passed to Syntax Processor. Syntax Processor interpreted the digital information and generates DFD Layout Structure. Display Builder gets the layout structure, it restructures the diagram and generates display information, while Semantic Processor takes in the layout structure and produces DFD Logical Structure. Interpreter then processes the logical structure and provides a set of analysis services for the system represented by the DFD.

## 7.2 Design Refinement

As illustrated in Figure 7.1–1, the entire system consists of seven components. They are: Scanner, Syntax Processor, Display Constructor, Semantic Processor, Interpreter, Service Builder, and a graphic user interface.

### 7.2.1 Graphic User Interface

GUI serves as a system manager, which launches all kinds of services including DFD analysis service, edit service, help service etc. It is can be decomposed into six interfaces shown in Figure 7.2.1–1.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

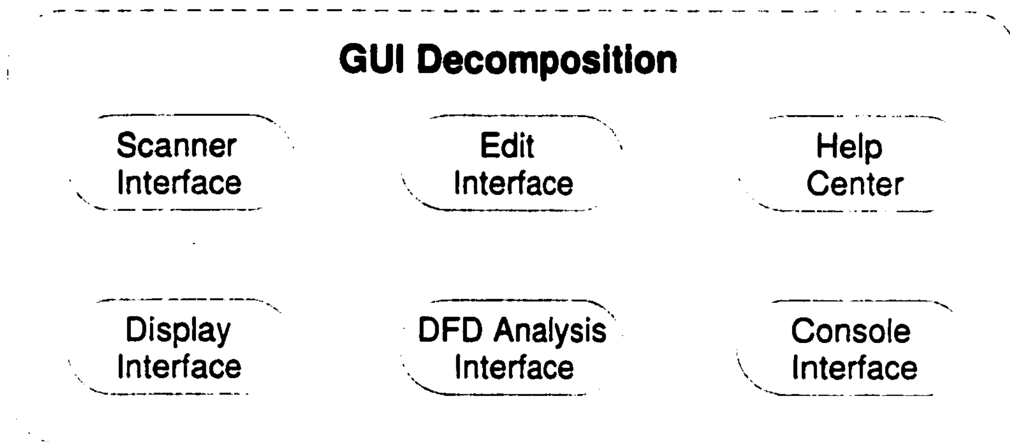



Figure 7.2.1-1

### 7.2.2 Syntax Processor

Syntax Processor takes in the digital representation of a DFD, extracts DFD features, analyzes the data, and generates DFD layout structure. Its components are shown in Figure 7.2.2–1

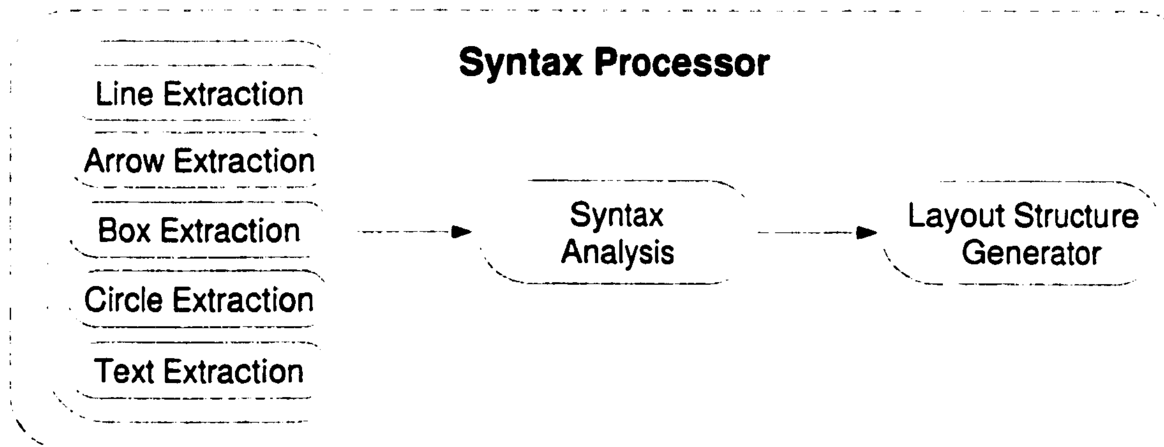



Figure 7.2.2-1



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

### 7.2.3 Semantic Processor

Semantic Processor takes DFD layout structure as input, checks the syntax, decomposes it into levelled DFD representation, analyzes the semantics, interprets it, and generates DFD logical structure. The decomposition of the semantic processor is shown in Figure 7.2.3–1

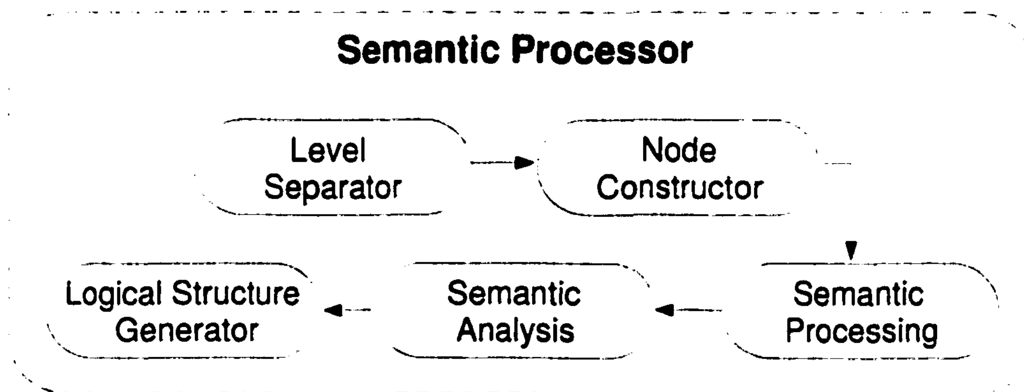



Figure 7.2.3-1

### 7.2.4 Service Builder

Service Builder matches the DFD display layout, traces transition between states, provides a set of DFD analysis services includes simulation, equivalence checking, and deadlock detection etc. The breakdown of the service builder is shown in Figure 7.2.4–1.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

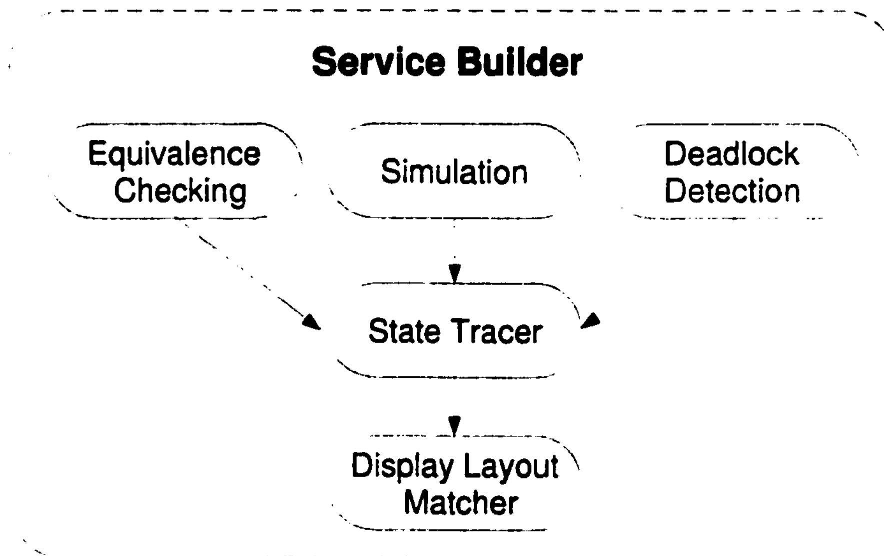



Figure 7.2.4-1

### 7.2.5 Display Constructor

Display Constructor no more than an automatic graphic drawing tool. Based on DFD layout structure, it extracts every entity, designs display layout, optimizes the display structure, and draws diagrams. Its components are shown in Figure 7.2.5–1



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

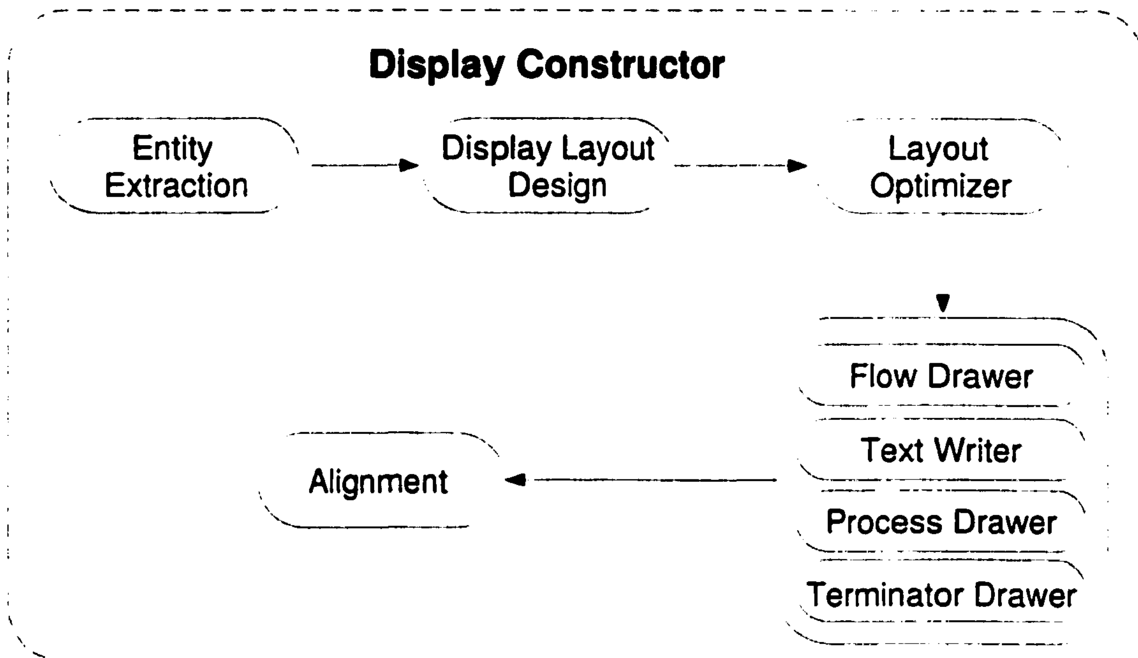



Figure 7.2.5-1

### 7.2.6 Interpreter

Interpreter will interprets the DFD logical structure and provides semantic meaning for variety of system analysis services. Such an interpreter can be directly used by importing CWB tool.

### 7.2.7 Scanner

Quite a few choices of diagram scanner are available in the market. This is can be done also by direct importation.

## 7.3 Simulation Sub-System

From what illustrated above, we can see that the entire system design and implementation needs substantial amount of time. The complexity of the entire



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

system exceeds the scope of a master thesis. However, a subset of the system can be implemented to demonstrate the underlined theoretical basis, which describes the semantic representation of DFDs.

### 7.3.1 Simulation Sub-System Design

The logical structure of such the simulation sub-system is shown in Figure 7.3–1. GUI is the system administration manager that can open a text based DFD logical structure, retrieves DFD display layout, displays the DFD in the GUI window, invokes Interpreter to precess the semantic representation of DFD, and provides behavior analysis for the DFDs.

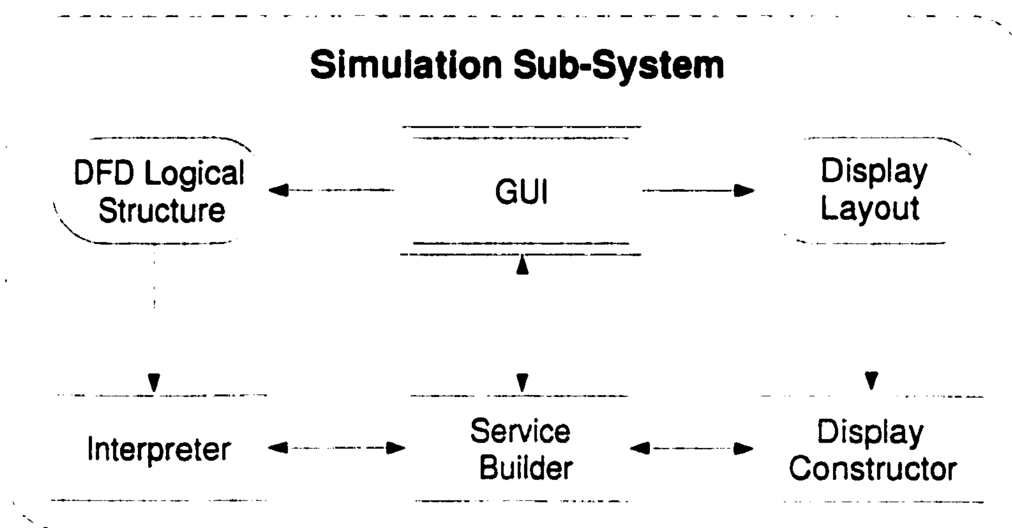



Figure 7.3.1-1

### 7.3.2 Assumption

The above simulation sub-system design is based on the following assumptions:



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1. The DFD layout structure is already processed by some of the components of Display Constructor such as components Entity Extraction, Display Layout Design, Layout Optimizer as shown in Figure 7.2.5–1.
2. An optimized DFD display layout is ready for Display Constructor to draw boxes, lines, circles, arrows, and to write text in the GUI window.
3. The DFD logical structure is ready for interpreter to process.
4. Concurrency Workbench is directly used as the interpreter.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 8 SYSTEM IMPLEMENTATION

The implementation mainly involves graphic user interface and simulator. An user interacts with the GUI to invoke the simulator for the simulation of a system behavior through its DFDs description.

## 8.1 Class Design

The demo system class design is originated from object-oriented strategy by following top-down approach. In terms of Demo System Design shown in Figure 7.3.1–1, eleven classes and a connection component are developed. CWB is imported as the interpreter.

### 8.1.1 Class Dependence Structure

According to their functionality, the classes can be divided into three levels. The first level is the program driver and a graphic user interface. The second one is the services the demo system provides. The third one is function classes that support the services. Figure 8.1.1–1 shows the class dependence structure.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

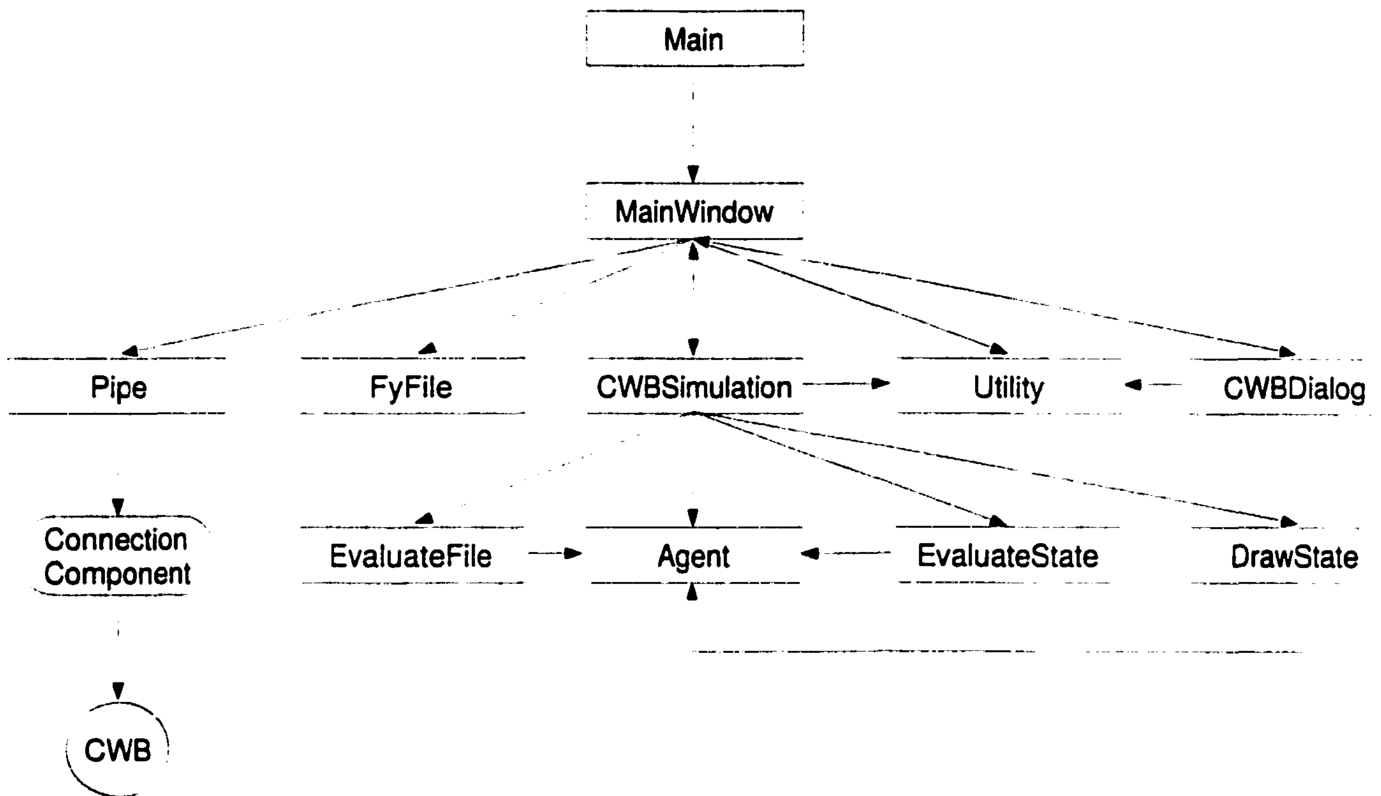



Figure 8.1.1-1

### 8.1.2 Class Specification

Class specification gives a brief description about the class and lists only major attributes and methods. The convention used in describing the classes is as following:

1. Attribute is described by name and type. The format is "+name: type". "+" or "-" sign stands for public or private.
2. Operation is described by name, parameter, and return type. The format is "+name (parameter list): return type". "+" or "-" sign stands for public or private.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**Main** is the program driver that will start the graphic user interface.

Operation

1. +main (String)

**MainWindow** is the graphic user interface that allows a user to interact with the system through mouse click. It includes a menu bar with four menus: File, CWB, Font, Background. Font and Background menu provide window property configuration service. File menu provides load file and quit system service. CWB menu provides a subset of CWB services that include: simulation, equivalence checking, difference checking, system size, states, minimum space.

Attribute

1. −myFiles: MyFiles
2. −cwbDialog: CWBDialog
3. −cwbSim: CWBSimulation
4. −utility: Utility
5. −font: Font
6. −fontName: String
7. −fontStyle: int
8. −fontSize: int
9. −foreground: Color
10. −background: Color
11. +exchange: Pipe
12. +cwbResponse: String
13. +invokeCWB: boolean
14. +loadFile: boolean



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

15. +buffer: byteArray

16. +bufferSize: int

Constructor

1. +MainWindow ()

Operation

1. +action (Event, Object): boolean
2. +handleEvent (Event): boolean
3. +paint (Graphics)
4. −fileAction (Event)
5. −cwbAction (Event)

**MyFiles** loads into the main window the DFD files that are written in CWB notation and DFD layout structure.

Constructor

1. +MyFiles (Frame, int)

**Pipe** is the port to connect imported CWB tool.

Operation

1. +setPipe ()
2. +read (): String
3. +write (String)



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

**CWBSimulation** is the simulation interface that allows a user to simulate the system behavior represented by DFD through mouse click. It contains normal window property setting options and methods to display simulation diagram in the simulation window dynamically.

Attribute

1. –mainWindow: MainWindow
2. –evaluateFile: EvaluateFile
3. –evaluateState: EvaluateState
4. –drawState: DrawState
5. –agent: Agent
6. –utility: Utility
7. –idle: String
8. –ready: String
9. –active: String
10. –Source: String
11. –Sink: String
12. –Process: String
13. –fileContent: String
14. –numberOfAgent: int
15. –systemState: charArray
16. –cwbResponse: String
17. –transition: charArray
18. –numOfTransition: int
19. –agentIndex: int
20. –button: Button



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

21. –font: Font
22. –fontName: String
23. –fontStyle: int
24. –fontSize: int
25. –foreground: Color
26. –background: Color

Constructor

1. +CWBSimulation (Frame)

Operation

1. +action (Event, Object): boolean
2. +handleEvent (Event): boolean
3. +mouseDown (Event, int, int): boolean
4. +paint (Graphics)
5. –paintAndReturn (): boolean
6. –setTransitions (int)
7. –setClicks ()
8. –setStates ()
9. –traceBack (int)
10. –traceForword (int, int)

**CWBDialog** is a dialog box served as an interface to CWB. It allows a user to issue CWB command to perform CWB operations through a input command text field. The CWB response will be displayed in the box window.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Attribute

1. −mainWindow: MainWindow
2. −utility: Utility
3. −button: Button
4. −textField: TextField
5. −font: Font
6. −fontName: String
7. −fontStyle: int
8. −fontSize: int
9. −foreground: Color
10. −background: Color

Constructor

1. +CWBDialog (Frame)

Operation

1. +action (Event, Object): boolean
2. +handleEvent (Event): boolean
3. +paint (Graphics)

**Utility** is the window display utility class served as an interface to manipulate window display properties such as foreground, background, font, font color, font style, font size, text fields, buttons.

Operation

1. +fileMenu (Menu, MenuBar)
2. +fontMenu (Menu, MenuBar)



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3. +backgroundMenu (Menu, MenuBar)
4. +cwbMenu (Menu, MenuBar)
5. +customPrint (String, Graphics)
6. +fontNameAction (Event, String): String
7. +fontSizeAction (Event, int): int
8. +fontStyleAction (Event, int): int
9. +foregroundAction (Event, Color): Color
10. +backgroundAction (Event, Color): Color

**Agent** is the class that store the information about agent such as name, type (source, sink, process), state (idle, ready, active), and position in the display window; about flows such as input flows, output flows, flow state (idle, ready, active), position; and about label position.

Attribute

1. +name: String
2. +type: String
3. +agentState: String
4. +agentIcon: intArray
5. +agentLabel: intArray
6. +inflow: charArray
7. +outflow: charArray
8. +inflowState: charArray
9. +outflowState: charArray
10. +outflowIcon: intArray
11. +outLabel: intArray



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

12. +arrow: intArray

Constructor

1. +Aent (int, int, String)

Operation

1. +getOutLabel (int): String

**EvaluateFile** is the system input file evaluation interface with all finds of file operations in it. It evaluates DFD display layout structures. It sets the coordinators of processes, sources, sinks, lines, arrows, text in agent object. It decides how many inflows and outflows associated an agent. It also initial agent states. flow states, and system states.

Attribute

1. −idle: String
2. −ready: String
3. −active: String
4. −Source: String
5. −Sink: String
6. −Process: String

Operation

1. +getNumberOfAgent (String): int
2. +createAgents (String, Agent, charArray, int)
3. +initialAgents (String, Agent, int)
4. +setIcons (String, Agent)



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. –createAgent (String, Agent, int)
6. –getNumOfFlow (intArray, String)
7. –createSystemState (String, charArray)
8. –initAgent (String, Agent, int)
9. –setAgents (String, Agent, int)
10. –initAgentState (String, Agent, int)
11. –initFlowState (Agent, int)
12. –insertFlows (String, Agent, int)
13. –insertInflows (Agent, int, String)
14. –insetOutFlow (Agent, int, String)
15. –setIcon (String, Agent)
16. –setOutflowIcon (Agent, StringTokenizer, int, int)

**EvaluateState** is the interface to update current and previous system states in terms of CWB response. It gets the number of possible transitions, updates transition array, finds out the agent and flow that were clicked in simulation window.

Attribute

1. –idle: String
2. –ready: String
3. –active: String
4. –Source: String
5. –Sink: String
6. –Process: String



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Operation

1. +setSystemStates (String, String)
2. +clickAgent (Agent, int, int, int): int
3. +clickFlow (Agent, int, intArray, int, int): boolean
4. +getTransitions (String, charArray, int)
5. −insetTransition (StringTokenizer, charArray,int)

**DrawState** is the interface to draw diagram in simulation window according to agent states.

Attribute

1. −idle: String
2. −ready: String
3. −active: String
4. −Source: String
5. −Sink: String
6. −Process: String

Operation

1. +setState (Graphics, Agent, int)
2. −setTerminator (Agent, Graphics)
3. −setProcess (Agent, Graphics)
4. −setFlow (Agent, Graphics)
5. −drawTerminator (Agent, Color, Graphics)
6. −drawProcess (Agent, Color, Graphics)
7. −drawFlow (intArray, Color, Graphics



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8. −drawArrow (intArray, Color, Graphics

9. −printAgentLabel (Agent, Color, Graphics)

10. −printFlowLabel (Agent, Color, Graphics)

## 8.2 Concurrent Processes Management

Since the interpreter to be used in demo system is imported from CWB, it has to be seamlessly integrated into the main program. Consider a port is built with the main program, a different application can be plugged in it such that the main program may interact with the plug-in application as though the application run stand-alone. The basic concept for such kind of integration comes from different process running independently but with the mechanics to communicate each other. This concurrent process creation, communication establishment, plug-in application involves quite a few steps.

### 8.2.1 Concurrent Process Creation

In order to run imported CWB, a concurrent process must be created as CWB bearer. This is can be done by using unix system call *fork* as shown in Figure 8.2.1−1. The Parent process and the child process are running concurrently but independently of each other.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

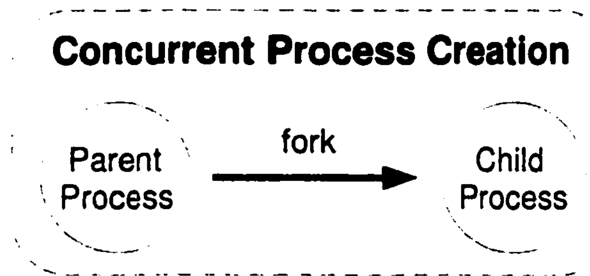



Figure 8.2.1-1

## 8.2.2 Communication

The two way real-time communication is then established through unix system call *pipe* and *dup* as shown in Figure 8.2.2–1. Two pipes are set up, one for read and one for write. Now the two processes are capable of talking each other in simplex mode.

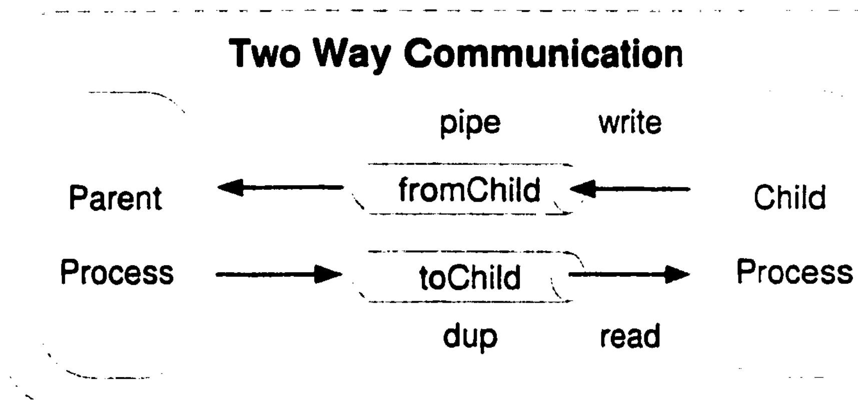



Figure 8.2.2-1



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## 8.2.3 CWB Invocation

When the concurrent process environment is set up, CWB is invoked by unix system call execlp and the two pipes are opened by system call fdopen as shown in Figure 8.2.3–1

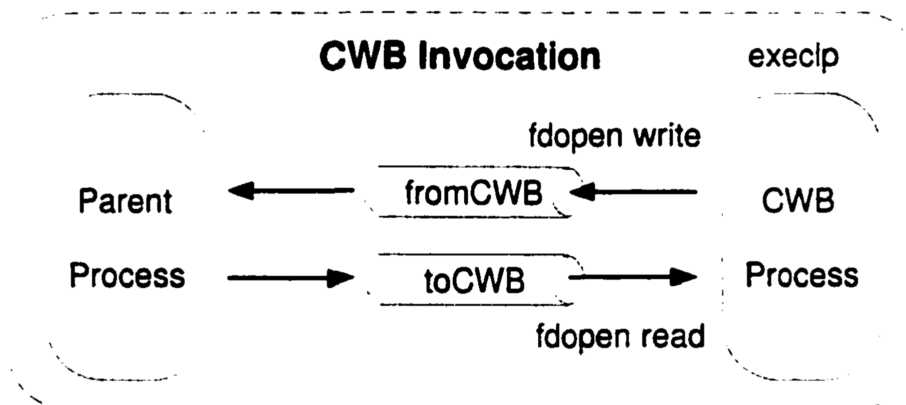



Figure 8.2.3-1

## 8.2.4 Main Program Linking with CWB

The last step is to link the main program with the CWB process after all the above preparations are done. Since the program handle the CWB process is written in C while the main program is written in Java. There must be a port in main program to allow CWB application plugged in. This can be done through the advanced Java technique called **native method.**

There are three native methods included in the Java port class Pipe that is served a an interface to other application implemented in different language. They are setPipe method, read method and write method. They are abstract method declared in Pipe class. The implementation of these native methods is accomplished by C code. The setPipe method is implemented in C as setPipe function that



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

establishes the concurrent process environment, invokes CWB application, and opens two way communications. The write method is implemented in C as write function that issues CWB command, converts Java string into C string, and writes it into toCWB pipe. The read method is implemented in C as read function that retrieves CWB response from fromCWB pipe, converts C string into Java string, and returns response to main program.

Java native method builder is used to create the middleware that match Java methods with corresponding C functions. C compiler library option is used to generate a shared library to be accessed by both Java methods and corresponding C functions. Figure 8.2.4–1 shows the described linking approach.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

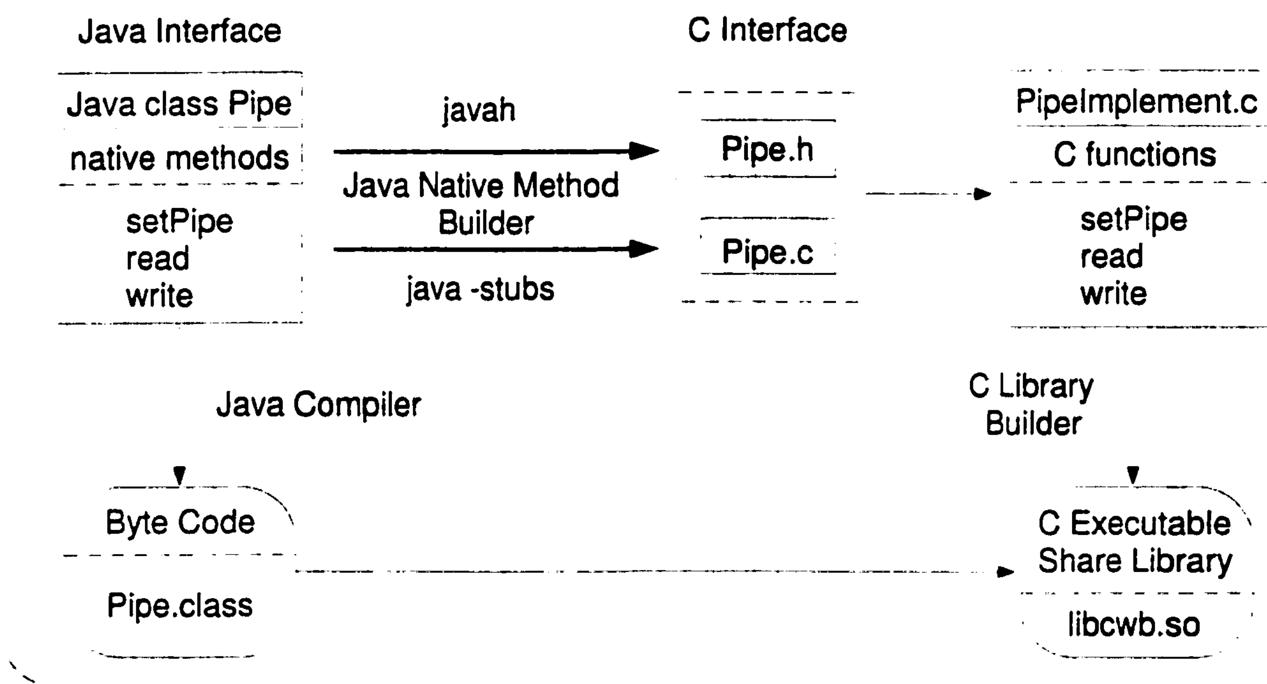



Figure 8.2.4-1

## 8.3 Programming Languages

Java and C are the two programming languages to be used for implementation. There are a number of reasons to choose Java.

1. Java is a pure objected-oriented language. Since object modelling technique is used in design stage, implementation in object-oriented language is more natural and compatible with design model. It will be easier to transform design model into implementation model.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. Java is an advanced modern language. Its specifications included the latest programming technique. It has a lot handy and powerful features such as graphic interface builder, string manipulation that save programmer substantial time.
3. Run time security checking, automatic garbage collection, and reference passing mechanism reduce program crash possibility and make the language more reliable.
4. The most import advantage of Java is expressed by its logan "write once, run anywhere". A platform independent, reusable software has long been the goal of programming language. Java archives this goal to great extent. It represents the future of programming language.
5. Another prominent feature of Java is its applet. Applet allows remote execution. As internet is exploring, Java becomes hottest technique in internet application development.
6. Java has a built in feature to allow plug in applications written in different language. This native method is perhaps played a key role in my implementation model.

## 8.4 Environment

The requirement for current implementation model is pretty simple. It only requires Unix system VI above with JDK1.0, and CWB installed. 55K source files are currently reside in schoenfinkel under /home/ucc/disk004/tzhou/thesis/implemt/java/interface1. The byte-code and shared library of about the same size are also installed in the directory. By typing java Main under this path, we can run the demo system with a nice interface.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 9 FURTHER WORK

There could be a lot of further work to do on this research topic. From what I consider needs substantial effort, the further work can be divided into three portion: theoretical portion, design portion and implementation portion.

## 9.1 Theory Work

The two key theory issues: DFD layout structure and DFD logical structure are pretty much done. Another important issue less concerning theory but more design is to establish a foundation for representing levelled dataflow diagrams.

## 9.2 Design Work

In section 7, we omitted the interpretation of levelled DFDs at both high level phase and refined phase since we did not discuss how to handle the levelled DFDs in DFD representation portion. This could result in adding a couple of more components in design and in restructuring the design diagram.

## 9.3 Implementation Work

Even through some of components could be imported into the system directly from commercial products, these products may need to be customized to fit the system requirement. Other components not implemented in simulation sub-system of course need substantial effort such as display layout construction.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

# 10 CONCLUSION

Formal specification provides a valuable approach to develope a powerful CASE tool which can semantically understand a system modelled by DFDs. This tool can be used to simulate a system behavior, check equivalence of two systems and detect possible deadlock. These features grant the tool usefulness in every phase through entire software life cycle. The architecture of the tool is based on a platform independent foundation, which makes it capable of doing system analysis both for new system design and legacy system migration at high level.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## BIBLIOGRAPHY

[1] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, pages 36–48, July 1989.

[2] G. Butler, P. Grogono, R. Shinghal, and I. Tjandra. Retrieving information from data flow diagrams. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 22–31, Toronto, Ont., July 1995.

[3] G. Butler, P. Grogono, and I. Tjandra. Analyzing the logical structure of data flow diagrams in software documents. *Document Analysis and Recognition*, pages 54–58, 1995.

[4] M-J Chen and C-G Chung. Preventive structural analysis of dataflow diagrams. *Information and Software Technology*, 34(2):117–130, Feb. 1992.

[5] Ming-Jie Chen and Chyan-Goei Chung. On the design of FLEDGED — a flexible editing tool for data flow diagrams. In *Proceedings of 3rd International Conference on Software Engineering and Knowledge Engineering*, pages 285–290, Skokie, IL, June 1991.

[6] Tom DeMarco. *Structured Analysis and System Specification*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1979.

[7] Thomas W. G. Docker and Graham Tate. Executable data flow diagrams. In D. Barnes and P. Brown, editors, *Proceedings of the BCS/IEE Conference 'Software Engineering 86'*, pages 352–370, London, UK. 1986. Peter Peregrinus Ltd.

[8] R. B. France. Semantically extended data flow diagrams: A formal specification tool. *IEEE Transactions on Software Engineering*, 18(4):329–346, April 1992.

[9] M. D. Fraser, K. Kumar, and V. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Transactions on Software Engineering*, 17(5):454–465, May 1991.

[10] A. Fuggetta. A classification of CASE technology. *Computer*. pages 25–38, Dec. 1993.

[11] A. Fuggetta, C Ghezzi, D. Mandrioli, and A. Morzenti. Executable specifications with data-flow diagrams. *Software — Practice and Experience*, 23(6):629–653, June 1993.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[12] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1979.

[13] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1991.

[14] S. Hekmatpour and M. Woodman. Formal specification of graphical notations and graphical software tools. In *Proceedings of 1st European Software Engineering Conference*, pages 297–305, Stradbourg, France, Sept. 1987.

[15] Charles F. Martin. Second-generation CASE tools: A challenge to vendors. *IEEE Software*, pages 46–49, March, 1988.

[16] James Martin and Carma Mcclure. *Software Maintenance: The Problem and Its solution*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1983.

[17] C. Mcclure. *CASE Is Software Automation*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1989.

[18] Jr. R. N. R Meeson, M. B. Dillencourt, and A. M. Rogerson. Executable data flow diagrams. In *CASE 87 — First International Workshop in Computer-Aided Software Engineering*, pages 445–454, Cambridge, MA, 1987.

[19] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–561, June 1995.

[20] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1989.

[21] H. S. Modell. More CASE on the Mac: Turbo CASE and MacBubbles. *IEEE Software*, pages 133–135, Jan. 1990.

[22] P. Newcomb and P. Martens. Reengineering procedural into data flow programs. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 32–38, Toronto, Ont., July 1995.

[23] Laurence Peters. *Advanced Structured Analysis and Design*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1987.

[24] L. B. Protsko, P. G. Sorenson, and J. P. Tremblay. Mondrian: System for automatic generation of dataflow diagrams. *Information and Software Technology*, 31(9):456–471, Nov. 1989.

[25] E. L. Reilly and J. W. Brackett. An experimental system for executing real-time structured analysis models. In *Proceedings of 12th Structured Methods Conference*, pages 301–313, Chicago, IL, 1987.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[26] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1991.

[27] S. R. Schach. *Software Engineering*. Aksen Associates Incorporated Publishers, Boston, MA, 1993.

[28] L. P. Tan, T. S. Chua, and P. T. Lee. AUTO-DFD: An intelligent data flow processor. *The Computer Journal*, 32(3):194–101, 1990.

[29] J. R. Tirso and H. Gregorius. Information reuse parallels software reuse. *IBM Systems Journal*, 32(4):615–620, 1993.

[30] T. H. Tse and L. Pong. Towards a formal foundation for DeMarco data flow diagrams. *The Computer Journal*, 32(10:1–12, 1989.

[31] David Walker. Introduction to a calculus of communicating systems. *Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh*, Feb. 1987.

[32] P. T. Ward. The transformation schema: An extension of the data flow diagram to represent control and timing. *IEEE Transactions on Software Engineering*, SE-12(2):198–210, Feb. 1986.

[33] Mike Webb and Paul Ward. Executable data flow diagrams: An experimental implementation. In *Structured Development Forum*, pages 1–21, Seattle, WA, 1986.

[34] R. Welland, S. Beer, and I. Sommerville. Method rule checking in a generic design editing system. *Software Engineering Journal*, pages 105–115, March, 1990.

[35] M. Woodman. Yourdon dataflow diagrams: A tool for disciplined requirements analysis. *Information and Software Technology*, 30:515–533, Nov. 1988.

[36] S. Yang. Two CASE tools for the macintosh. *IEEE Software*, pages 120–123, Jan. 1989.

[37] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1989.

[38] E. Yourdon. What ever happened to structured analysis. *Datamation*, pages 133–138, June 1986.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[39] L. L. Yourdon, E. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1979.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

## VITA AUCTORIS

**Lizhong Zhou** was born in 1954 in**Beijing, China**. He graduated from**Beijing Normal University** with B. Sc in Electrical Engineering in 1982. From there he went on to the**Beijing Institute of Technologies** where he obtained a M. Eng. in Industrial Engineering in 1989. He was back to school as a candidate for the Master's degree in Computer Science at the University of Windsor in 1995 and graduate with M. Sc. in 1998. He is currently working in telecommunication industry as a software engineer.



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.