A Prototype of an ABL Syntax-Driven Editor Supporting Software Development

Kenneth Finkelstein

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

December 1988

ISBN 0-315-51354-3

ABSTRACT

A Prototype of an ABL Syntax-Driven Editor
Supporting Software Development

Kenneth Finkelstein

A new set of ABL (Alternative Based Language) tools has been designed which addresses the problems in the existing ABL implementations, in particular with respect to the user-interface. The ABL methodology was introduced by W. M. Jaworski as a way to deal with the crisis in software development.

This crisis had arisen because software systems were getting increasingly large, and the previous techniques that had been used for software development were just not able to meet the demands created by these new large software systems. In order to make large software systems more manageable, the process of designing, implementing, and maintaining them was broken down into discrete parts, which were collectively referred to as the software lifecycle.

Using ABL tools, a number of software projects have been designed and implemented, and the people involved have felt that though there were problems with the ABL tools, the ABL

methodology had been instrumental in the success of these software projects.

The use of a syntax-driven editor goes a long way towards creating an environment in which software systems can be built without the use of paper and pencil. Its full screen editing facilities, and quick response time, make it possible to design software systems in a natural, intuitive fashion.

Acknowledgements

I would like to thank Professor W. M. Jaworski for giving me the opportunity to work with the ABL methodology professionally.

I would like to thank Professor J. Opatrny for helping make this thesis a reality. It would not have been completed without his guidance.

I would like to also thank Greta Nemiroff for teaching me that it is alright to succeed, and for being such a good friend.

I would like to thank my parents for their help and understanding, and finally, I would like to thank Leona Heillig, who has had to put up with me as a student for almost nine years. Let us hope that we will have more free time together over the next nine.

TABLE OF CONTENTS

Introduction

The first chapter explores the roots of software engineering, from the reasons why techniques for building small software systems are not adequate for building large software systems, to the defining of the stages of the software lifecycle, as a means of having guidelines in building large systems, (which are both efficient, reliable, and maintainable).

An informal introduction to the ABL methodology is included in the second chapter. This is followed by a discussion which shows some of the strengths of the ABL methodology, and the benefits of designing and implementing software systems using the ABL methodology.

The third chapter is devoted to presenting some of the software systems that have been built using the existing ABL tools. An attempt has been made to choose software projects which would illustrate the different types of applications that the ABL methodology is best suited to. These applications fall into the categories of software automation, software salvaging, and real-time process control.

The first three chapters serve to lay the groundwork for the remaining chapters, which concern themselves with the implementations of the ABL methodology. Though the ABL methodology has been recognized as a valid method for designing and implementing large software systems, there has been sharp criticism of the ABL tools, particularly in the area of the user-interface. What is being presented here is a prototype for a new set of ABL tools, which remedy the major problems in the existing ABL tools, by changing the user-interface from a hierarchical menu-driven data-entry system to a syntax-driven full screen editor. Chapters four through six address themselves to the various aspects of the ABL tools, contrasting the existing implementations with the prototype for the new ABL tools.

Chapter four examines the user-interface of each set of ABL tools, with the emphasis placed on the editing and navigational facilities. The user-interface should allow the user to enter ABL processes in a natural, intuitive fashion. It should also make sure that the ABL methodology is adhered to (in a way that is unobtrusive to the user). Chapter four pinpoints where the user-interfaces succeed and fail on these points.

Chapter five discusses the choices that were made in the design of the internal structure in all versions of the

ABL tools, and considers such implementation details as the choice of computer language.

Chapter six looks at the underlying database found in all versions of the ABL tools. The benefits of a database are expounded on, and the principle database tables are presented.

The seventh chapter is divided into two sections. The first section lists the functions that the prototype contains and discusses wha' features must be added before the new tools can be used in industry. The second section indicates the possibilities for future research. Though the existing design for the ABL tools is sufficient for designing and implementing large software systems, there is still room for improvement. This section examines some of the places where the ABL tools can be improved, and discusses some of the possible ways that these improvements can be made.

CHAPTER 1

AN INTRODUCTION TO SOFTWARE ENGINEERING

In the early days of software development, machines were relatively small and so were software systems. A programme usually existed in a small, well-understood environment or domain, and could be written directly from a statement of need. Normally, this programme was written by one individual, and it was not uncommon for most of the design and implementation details to be kept in the programmer's head.

The programmer would usually programme with respect to a particular machine, and the mark of a skilled programmer became the ability for the programmer to find a way to save a machine cycle, or reduce the overall size of a programme by several lines, (not unlike the concept of an "elegant" proof in symbolic logic), usually at the expense of the understandability of the code. The importance of good documentation was not appreciated, so there was no real attempt to describe how a programme worked.

At the time, (late 50's till mid 60's), this method for programming had some validity. Computer time was very expensive and programmer time was not, and the memory of computers was small, so the preoccupation with saving time and space when writing a programme was understandable.

With the advent of third generation computer hardware large applications that had been previously unrealistic became feasible. The implementation of these applications required large software systems to be built, and it soon became clear that there were major inherent flaws in current programming design techniques. The advances that had been made with computer hardware were not being mirrored by computer software. Sharon [33] (software design environments marketing manager at Tektronix), notes that computer speed and power are increasing by about 30 percent per year, while the productivity of the typical software developer is increasing by only about 4 to 7 percent per year. "The current computer and software-development situation is like building bigger, more powerful cars while the world is running out of gas."

It was not just new software systems that were being effected by this gap between hardware and software development. As computer hardware became more advanced, maintaining existing software systems involved transferring them to new hardware. Because the software systems were not designed properly this operation was very difficult, and sometimes required redesigning and implementing the existing software system.

The problems encountered in designing and implementing large software systems are not simply scaled up versions of the problems involved in writing small computer programmes. The use of more structured high-level programming languages helped, but did not alter the fundamental weaknesses of the conventional process used for developing software systems. Programming languages could not be the only formal tool in the development, validation, and maintenance of software.

The new large scale projects tended to be inundated with design and implementation problems. The schedules for these projects were unpredictable: a number of the projects in the late sixties and early seventies were late, sometimes by years, which led, naturally enough, to excessive costs. When the project was finally completed it tended to perform poorly, quite often was unreliable, and was difficult to maintain. In order to narrow the gap between software needs, (that have arisen from the increased power of computers), and software production, new methodologies and techniques had to be developed that are fast, flexible, and allowed large software systems to be built without running into the aforementioned problems (which collectively became known as the 'software crisis').

Software engineering is a discipline that has emerged as a means to deal with this 'software crisis'. Varying definitions have been given for this discipline, but the underlying theme behind all of them is its concern in building software systems that are complex, (at least beyond the scope of one individual), where the most important attributes of the software system are reliability, understandability, and maintainability. To this end, the concept of 'the software lifecycle' was introduced.

There are five distinct stages that comprise the period of development and usage known as the software life cycle. The first stage is the specification stage. It is here that the system constrai .s and the user requirements of the system are specified. The second stage is the design stage. Given the specification, it is at this stage that the question of "how" is addressed. The third stage is the implementation stage. The coding is done at this stage. The fourth stage is the testing stage. Here the implementation is tested to ensure that it meets the requirements and constraints of the specification. Finally there is the maintenance stage. This is where the system is in operation, and any day to day events, (such as backups), o. major modifications are performed.

1.1 THE SPECIFICATION STAGE

The importance of the specification stage cannot be minimized. As systems get more complicated it gets increasingly harder to understand them. In some large systems up to 95% of the code had to be rewritten to satisfy user requirements and also 12% of the errors discovered in a software system over a three year period were due to errors in the original system requirements [31].

The purpose of the specification stage is twofold. First it gives the designer, (or system engineer), a kind of buffer zone. Given how complicated a software system can be, the specification gives the designer a concrete map of the system to work with, instead of just having to start directly with the design stage.

The second reason for the specification stage is to serve as a means of developing a dialogue between the user, (or customer), and the system engineer. This transfer of knowledge is very important because quite often the system engineer knows little about the application, and the user knows little about software systems.

There are two classes of information that should be included in the specification document. The first class of information describes the functionality of the system, while the second class of information details restrictions imposed by the users.

1.2. THE DESIGN STAGE

The design stage details how to accomplish specifications defined in the specification stage. The design is arrived at by analyzing the software requirements contained in the specification document.

Since the design stage serves as the link between the user specification and the implementation, it can be seen as the most critical stage of the software lifecycle. How well the design reflects the user specification, how straightforward the design is to implement, and how easy the design is to understand, will be the factors that ultimately decide how reliable and maintainable the system will be.

1.3. THE IMPLEMENTATION STAGE

The third stage of the software lifecycle is the implementation stage, where the design of the software system is realized. There are certain guidelines that should be followed when implementing a software system. The guidelines cover such topics as the use of comments; constants, types and variables; portable software; the use of gotos; and testing. A discussion of these guidelines is better suited to a critique on programming style, and there are many books available on the subject.

1.4. THE TESTING STAGE

Calling the fourth stage the testing stage is a bit misleading, for it conjures up images of people running test data and debugging the system. Actually, what is being done here is more akin to an acceptance test. The customer or user is examining the software system to ensure that it meets the requirements and constraints of the original specification.

There should actually be communication between the customer and the system engineer from the design stage onwards. Waiting till the end of the project can be catastrophic in the case of either designing a system based on incorrect specifications or designing a system where the specifications had been misinterpreted.

1.5. THE MAINTENANCE STAGE

The maintenance stage actually encompasses more than one would think. When we speak of maintenance with regard to something like an apartment building, for example, we are thinking in terms of repairs and upkeep. In the software lifecycle, maintenance has a much broader definition. Here, it is not just a matter of repairs. When we speak of maintenance with regard to software systems, we mean: to modify in the case of error OR in the case of having to modernize. It is possible that whole subsystems of the software system can become obsolete, perhaps due to new equipment being installed, and consequently the subsystem has to be completely rewritten. A significant and in most cases a majority of O&M, (operations and maintenance), costs are due to product improvement as opposed to error correction [36].

Because of the problems of maintenance, most of the total cost of a software system occurs after the system is delivered, and goes into debugging and maintenance. For example, the estimated development cost for the software in a U.S. Air Force F-16 jet fighter is 85 million dollars. Yet the Air Force expects to spend 250 million dollars maintaining that software over the jet's operational lifetime [33].

When we consider the fact that maintenance accounts for such a disproportionate amount of the costs of a software system, we can appreciate how even a small reduction in maintenance can dramatically improve the overall costs of the system. In order to reduce maintenance costs, however, we have to look to the other stages of the lifecycle, for it is only by designing and implementing a more reliable and more easily understood system that we can hope to cut maintenance time and therefore, maintenance costs. If the software quality assurance and operation/maintenance issues continue to be ignored during the development phase, a disproportionate amount of the life cycle cost will continue to be assessed for operation and maintenance [36]. The problem with this approach, is that in a world where the lowest bid often wins the contract, people are hesitant about applying rigorous methods that increase the costs of software development, even if any increased development costs directly result in a reduction in maintenance costs and therefore an overall cost reduction.

A basic theme common to all (phases of software engineering cycle) can be summarized as follows : (1) ensure that the performance or behaviour required by the preceding state is met, and (2) minimize the errors passed on the successor stage [36]. The software lifecycle can be viewed as a series of ordered modules, (where each stage is a module). Given a particular module, the starting point, i.e., the input to the module, should be the previous module's output, and the output of the module should be the next module's input.

The progression from one stage to another stage is usually from the general to the particular, (for example, the general design to particular details of implementation), and quite often, different languages are used at different stages of the software lifecycle. The ramifications of this is that a statement made at one stage can translate into multiple statements at another stage.

When maintaining the system, if it is necessary to make changes, then these changes should be first made at the design level. If changes are made to the current stage of the software cycle, without regard to the previous stages, then the changes that are made amount to patchwork and the system will no longer correspond to the original design.

CHAPTER 2

AN INTRODUCTION TO THE ABL METHODOLOGY

2.1. AN INFORMAL INTRODUCTION

ABL, (Alternative Based Language), which is also known as SOS, (Strategy Oriented Software), was designed by W. M. Jaworski, as a means of expressing algorithms [16]. It is a framework by which a finite sequence of definite steps, which may consist of one or more operations, can be carried out systematically, (in a defined order), to solve a given problem. The algorithm being expressed can pertain to anything from filling out tax forms to baking cakes. However, its usefulness lies in its ability to express algorithms that relate to software, both at the top level (system design), and at the detailed level (programmes).

An ABL process combines two separate components, a strategy, which can be thought of as a control flow model, and an environment, which can be thought of as a data flow model. This division is useful when implementing a software system because the components can be reused or shared by different ABL processes. For example, consider two ABL processes which open a file, create a data packet from the information found in the file, close the file, and send the data packet to another process. Both these ABL processes use the same decision logic or strategy. The only difference lies in the data object (in this case the different files) that is being operated on. Because of the separation of strategy and environment it is possible for both ABL processes to use the same underlying strategy (or environment).

An ABL process is made up of clusters, which is the only control flow construct supported by the ABL methodology. Every cluster has an ID number, and the clusters are numbered from one to n, where n is the total number of clusters in the ABL process. An ABL process starts with cluster one, and termination of the ABL process (if the ABL process terminates), is denoted by cluster zero.

The cluster construct is composed of alternatives which in turn are composed of two elemental units, the conditional, or boolean expression, and the assignment statement. When designing a programme, the decision points, (which are based on boolean expressions), can be seen as the interesting events that take place in a programme because it is here that the choice of what path is taken will be made.

When a decision point is encountered a mechanism is needed for deciding which path (or alternative) will be chosen. To this end, each alternative in a cluster starts with a guard. A guard is an unordered set of conditions. Given a cluster with several alternatives, the guards for the alternatives must be mutually exclusive as well as exhaustive. They must ensure that an alternative can be chosen, and at the same time, that only one alternative will be chosen.

Once an alternative has been chosen, a set of actions, known as the action flow, is executed or performed sequentially. In general, the action flow of an alternative should not contain any control flow statements. Every alternative has a next or target cluster ID, and when the action flow has been completed for an alternative, control passes to the cluster that is referenced by the next cluster ID.

To illustrate this, consider a cluster taken from an interactive ABL process which has the task of guessing a number by using a binary search. The range of the number has been given using the data objects TOP and BOTTOM and the midpoint of the range, which is used by the ABL process as the current guess, is defined as MIDPOINT. The user communicates with the ABL process by means of a data object called USER_ANSWER. It is assumed that all of these data objects have previously been initialized with valid values.

```
C2  Find User Number

    A2      Guess is too low
            IF USER_ANSWER is too low
              assign MIDPOINT + 1 to BOTTOM
              calculate new MIDPOINT
              display MIDPOINT as new guess
              get USER_ANSWER
                    NEXT C2

    A3      Guess is too high
            IF USER_ANSWER is too high
              assign MIDPOINT - 1 to TOP
              calculate new MIDPOINT
              display MIDPOINT as new guess
              get USER_ANSWER
                   NEXT C2

    A4      Guess is correct
            IF USER_ANSWER is correct
              display prompt for whether to guess again
              get USER_ANSWER
                   NEXT C3

    A5      Invalid response from the user
            IF   NOT USER_ANSWER is too low
           AND  NOT USER_ANSWER is too high
           AND  NOT USER_ANSWER is correct
              display prompt for re-entering message
              get USER_ANSWER
                   NEXT C2
```

The sample cluster has four alternatives (A2 - A5). The first three alternatives have guards with a single condition while the last alternative has a guard with three conditions. The guards are exhaustive, since all possibilities have been accounted for, and are also mutually exclusive, because only one of the cluster's alternatives will be chosen.

When a simple data object, like a string, is defined, certain operations have also been defined, either explicitly or implicitly. For example, it is possible to create a new string by concatenating two existing strings, and it is also possible to extract a substring from a string. A data abstraction is the definition of a data object coupled with all permissible (valid or legal) operations on that data object. An example of a data abstraction would be a stack combined with the operations push, pop, and initialize. An environment can be defined as a set of related data abstractions [16].

To summarize, an ABL process is defined in terms of an environment and a strategy. An environment is composed of data objects, actions, and conditions. A strategy is composed of clusters and alternatives. An ABL process maps the strategy onto the environment primarily through the use of guards and action flows.

So far, what has been shown is an informal introduction to the ABL methodology. For a more in-depth treatment complete with examples, see Jaworski [14][18][19], and also consult the appendix which contains the glossary of ABL terms. What follows is a discussion of the strengths of the ABL methodology in designing and implementing software systems, and how the ABL methodology relates to the software lifecycle.

2.2. BUILDING SOFTWARE SYSTEMS WITH THE ABL METHODOLOGY

The ABL methodology was developed by W. M. Jaworski, as an aid in designing and implementing large real time software systems, software automation systems, and as a tool for software salvaging. It allows the user to design the software system using a top down approach, a bottom up approach, or a combination of the two. The ABL methodology can be used to describe the top level functionality of a system, which can be followed by stepwise refinement and hierarchical structuring. It is also possible to start with the detailed specifications of the lower design levels and work upward. This is especially useful for software salvaging when there is inadequate documentation for the existing software system. Working at the level of the system's programmes, it is possible, using the ABL methodology, to work from programmes to the subsystems to the system, in order to extract the design from the existing system. The ABL methodology is capable of addressing all stages of development and implementation with a single integrated notation. Using the ABL methodology large software systems can be built that are reliable, understandable, and maintainable.

When designing software systems, the ABL methodology allows the user to divide the design into two distinct stages, where the first stage is machine and language independent (and can be thought of as the specification for the system), and the second stage is machine and language dependent (and can be thought of as the implementation of the system). The first stage permits specifications to be written in a natural language, but at the same time, imposes structure on the language.

One of the advantages of being able to design a software system using a language independent environment, is that implementation details, such as choice of computer language, do not have to be decided until later on. The specifications can be mapped to any computer language. This is not meant to imply that a system can be implemented using any computer language if the ABL methodology is used in creating the specifications. Different computer languages are appropriate for different applications. Furthermore, though programmers using the ABL methodology have the advantage of working with a detailed design that clearly outlines the control flow of each programme, a working knowledge of the computer language being used is still necessary.

Because the first stage is machine and language independent, it makes it possible for application experts and software experts to be able to communicate effectively - even though the application expert may not know anything about software systems, and the software expert may have little or no knowledge about the application. This is a very desirable feature when first specifying a software system. What makes this process so feasible is that the ABL methodology is so easy to learn. An application expert can be comfortable reading and writing using the ABL methodology after just a few hours. Moreover, once the first stage of the design has been worked out, the software experts can then create the second stage directly from it. One of the strongest features of the ABL methodology is that this process works in both directions. If an error is found in a programme, correcting the error automatically updates the machine and language independent stage, which results in a system that always corresponds to its specifications.

One of the reasons that people can communicate so effectively using the ABL methodology is that differe views can be generated for an ABL process. Some people may be more comfortable working with a view that resembles a computer programme, while other people might prefer something that uses graphs or decision tables.

The use of decision tables as one of the views available to ABL users is very important, because decision tables are more flexible than ordinary source code[4][22]. Using the ABL methodology, any code segment can be reduced to a tabular format. By allowing the code to be represented as a decision table, it is possible to produce logically correct code which can be checked for completeness and consistency. As well, it is possible to optimise the ABL process before, during and after code generation [23].

The ABL methodology is very useful for writing system level designs, but it is particularly strong at the detailed design level. Programmes that are written using the ABL methodology actually have the detailed design incorporated into them as comments. Examples of this can be found in the report appendix.

The value of having a methodology which guarantees that detailed comments (that always reflect the design), exist, should not be minimised, given that any ten lines of a computer programme can be mistaken for any other ten lines, and that a fundamental problem when building and maintaining a system is that the programmes usually do not match the documentation. It is common practice for programmers to document a programme after it is finished, because they feel that since the programme will probably be changed several times, the intermediate commenting will have been done for no reason. The problem with this approach is that it usually results in incomplete documentation, either because there is no time at the end of the project, or because, the programmers are already impatient to move onto something new, and so do not comment in great detail. Another problem with leaving the detailed documentation until the end of the project is that very often a programmer will not work on a project from beginning to end. If a new programmer starts working on a programme while it is in an intermediate phase it might be very difficult for the programmer to understand the programme. This is not a problem when the ABL methodology is used.

Aside from having detailed descriptions for every data object used by a programme, there are three types of comments that are automatically included in a programme that has been designed using the ABL methodology.

3.2.1. Statement of Purpose

The first type of comment included in a programme that has been implemented from an ABL design, acts as a header or statement of purpose. Its function is to describe, in approximately a sentence, what the function of the programme is.

3.2.2. Top-level Control Flow

The second type of comment that appears in an ABL programme shows the top-level view of the control flow. Without going into the details of the control flow, these comments show the algorithm or strategy that the programme has been based on. The main sections of the programme are outlined here.

3.2.3. Detailed Control Flow

Every section of the ABL programme that is outlined in the top-level control flow has two parts. The first part consists of the strategy for the section as it appears in the top-level view. This has been expanded to include the detailed design. The second part of the section is the actual code which corresponds to the detailed design.

The structure that the ABL methodology gives programmes is useful for both seasoned programmers and novices. The experienced programmer will be less likely to use tricks when implementing a design, and this will lead to more reliable and understandable programmes. At the same time, the ABL methodology is not too restrictive, and will not hinder the programmer's creativity.

The ABL methodology will not ensure that an unqualified programmer will suddenly produce quality programmes. Likewise, if the initial design is badly thought out, an ABL implementation of the system will not save the system. But it is still very useful for a novice to be given a framework in which to programme within, and the ABL methodology provides just such a framework.

CHAPTER 3

THE ABL METHODOLOGY IN INDUSTRY

While the ABL methodology has been in existence for a number of years, it has only been in the past three years that it has been applied to the specification and implementation of large software systems. Most of these projects have been carried out by a small, Montreal based, software company which decided to convert to the ABL methodology [15] in 1985. The company works on a contract basis with large corporations that need specialized software such as turnkey systems, or that require software consulting services. The software systems provided are for industrial applications and includes the categories of plant automation, process control, data acquisition systems, data communications, and software salvaging. The underlying theme common to all these categories is that they are real-time embedded systems.

The company has designed and implemented a number of software projects using ABL tools, and the people involved have felt that the ABL methodology had been instrumental in the success of these software projects. The projects have been implemented on different computers, (for example, a PDP-11, a VAX, a HP 1000, and an IBM AT), and the software for the projects have been written using different computer languages, (which include several dialects of Fortran, C, and Pascal). Not only have the computer environments been different from project to project, but the applications have been varied as well. Some of the projects that were designed and implemented, in part or in whole, using the ABL methodology include, the automation of a rail finishing mill, an earth station for a satellite, a system to monitor power lines, a system to monitor phone lines, and the salvaging of software from an outdated hardware environment, and its conversion to and implementation on a newer machine. A few of the projects which were completely designed and implemented using ABL tools are presented here. These projects help to illustrate the areas in which the ABL methodology has proven useful.

3.1. AUTOMATING A RAIL FINISHING MILL

A steel company required a newly built rail finishing mill to be automated [17]. The mill had to be able to operate a paperless tracking and recordkeeping system that would also provide long-term storage of rail characteristic data. All rails entering the rail finishing mill were given the highest classification, and as the rails journeyed through the mill, they were tested and subsequently re-classified according to defects and imperfections that were discovered. Testing was conducted manually and with the use of special machines. All information regarding the rails were sent to the MIS computer when the rails left the mill.

After the rails had cooled in a cooling tank, they were sent to a straightener before being inspected. It was at this point that the system became aware of the rails. The outside of the rails was inspected manually with the help of ultraviolet lighting, and the inside of the rails was tested for imperfections by an ultrasonic tester. Each rail was cut with special saws, sorted according to classification, and shipped out.

The software system was implemented on a PDP-11 computer (called the rail finishing computer), that ran under the RSX11M+ operating system, which had seven terminals and three printers connected to it. As well there were three VAX computers that the rail finishing computer communicated with using DECnet. There were also sensors and switches on the rail beds that were used to monitor and control the path that the rails took to traverse the mill. These sensors and switches were under the dominion of a PLC (programmable logic controller) which communicated with the PDP-11 through common memory.

Operators can read and update information regarding the rails using one of seven workstations. These workstations are capable of displaying quality control information, notifying the user of alarms and significant events, and supplying general reports on the status of the rails.

The first workstation in the rail finishing mill is the straightener. The straightener console displays data regarding how to adjust the straightener for the current rails. When the rails have passed through the straightener they are sent to the visual inspection workstation.

As well as examining the rails for defects, the operators at the visual inspection workstation are responsible for deciding if the current rails require more straightening. Normally the rails are routed to the ultrasonic tester workstation from here, but if the rails need further straightening they are routed to the hydraulic gag press workstation first. The rails that are sent to the hydraulic gag press workstation are re-straightened and sent to the ultrasonic workstation.

The ultrasonic tester workstation has a dedicated VAX which controls the ultrasonic tester machine. The machine checks the internal structure of the rails and the VAX computer passes the information to the rail finishing computer.

The rails have to be sorted according to the status contained in the rail finishing computer. However, it is possible that there are additional cuts that must be made to the rails (perhaps due to defects). If there are rails that have to be cut, they are automatically routed to the saw number 3 workstation. The rails are then routed to a sorting bed by the rail finishing computer. A loader is responsible for unloading rails from the sorting beds. This is where the rails leave the system.

This is one of the largest software projects, if not the largest, that the ABL methodology was used in. The project took almost two years to complete, from the specifications to the final acceptance tests, and involved approximately eight system engineers.

The ABL methodology was used to create a detailed and comprehensive specification which was developed in conjunction with the client. This specification became the contractual agreement between the company implementing the software system and the client. By designing and implementing the system using the ABL methodology, monthly progress reports were produced automatically and sent to the client.

Since every line of code was written using the ABL methodology, all the subsystem documentation and their corresponding programmes had a homogeneous look to them. Because of this, a system specialist who had designed and implemented one subsystem was able to, without much difficulty, understand another subsystem. This became very important during the commissioning stage (when the system was being installed and tested on-site), because it was not always possible to have everybody who was involved with the project on-site at the same time, owing to the fact that the rail mill was hundreds of miles away from Montreal.

3.2. A CIRCUIT BOARD TESTING SYSTEM

The need for software salvaging is becoming more prevalent in the computer industry. Computer systems are getting older, maintenance is becoming more expensive, especially in the case of systems that were not designed properly. Also, with the advances that have been made in computer hardware, existing systems quite often are unable to meet the current needs of users. One solution to these problems is to build new systems. However, in many instances the users are unwilling to give up their current system, because they are so dependent on them.

The idea behind software salvaging is to implement a software system which is functionally equivalent to the system that now exists. The new system should even faithfully reintroduce any bugs that might have existed in the original system. The changeover should be completely transparent to the user, although there might be slight differences for such things as backups, in the case where the software system is being transferred to a different type of computer.

One of the problems with software salvaging is that quite often there does not exist proper documentation for the existing system. In this case, the specifications for the system must be extracted from the existing code, in contrast to the usual system design, which involves working from the specifications to the code.

When extracting the design from the existing implementation there are two strategies that can be employed. The first is to just map the old code to the new code, and the second strategy is to try and redesign the new code so that it better implements the design which is extracted from the existing code. The second method is preferable because the result is a better defined software system, but in order to use this method, a complete understanding of the existing system is necessary. This is not trivial, especially when the existing code is written in one language and the new code is written in another language.

The circuit board testing system [29][30] was running on a PDP-9, only one of two such computers still operating in the world. All the programmes were written in assembler code. The software salvaging project consisted of converting this system to an HP-1000 computer with the programmes written in the C language. The existing system also used custom designed hardware which communicated with the PDP-9. The communication protocols had to be mapped to the new system also, so that existing test programmes would still function.

There were approximately 10,000 lines of assembler code in the existing system, and the only programmer's documentation was comments imbedded in the programmes. Aside from the difficulties of reading and understanding assembler code, and the problems inherent in translating a programme from one language to another, the word sizes of the two computers were not the same.

The first stage of the conversion process involved studying the assembler code in order to extract informal algorithms. Programmes were then grouped into a hierarchy of ABL processes, and the informal algorithms were converted to the narrative descriptions used in the ABL methodology. At this stage, some of the verification tests which checked for consistency and completeness were able to be conducted.

Once the strategies for the ABL processes were complete, C code was added to the environment, and then programmes were generated for each ABL process. The system was tested and debugged module by module, and in general, there were few problems.

This project took sixteen months to complete and involved approximately five system engineers. It was generally acknowledged that the ABL methodology was invaluable to the process of software conversion, particularly as an aid in arriving at the semantic understanding of an existing module. As well, by using the ABL methodology, a hardware independent design of the software system was produced.

3.3. THE TELECOMMANDE EN LIGNE PROJECT

When there is a break in a power line, it can take a great deal of time and effort to localize the break, particularly if the power lines effected are in a remote region. In order to minimize the time between becoming aware of a break and locating it, the Telecommande en Ligne project [9] was conceived. The Telecommande en Ligne network consists of master stations and remote stations, where the master stations monitor and control remote stations, which in turn are responsible for a certain geographical region. Within these regions, the power lines have been grouped into segments. By examining the segments in a region, it is possible to quickly determine where a break has occurred.

The Telecommande en Ligne project was broken into several projects which were designed and implemented by different companies over a two year period. The project dealing with the software system for the master stations was completely designed and implemented using the ABL methodology.

The Telecommande en ligne project is a multitasking system, running on a VAX computer under VMS version 4.4, in which there does not exist any central or controlling process. The system is divided into subsystems based on their functionality. The project consists of three main components, communication with the remote stations, a workstation interface for an operator, and maintenance of a database that is a repository for information concerning both the workstation itself, and information about the remote stations.

The subsystem responsible for communication with the remote stations transmits and synchronizes "telecommands" using polling and interrupts. Changes in the status of the remote stations are monitored, measurements from the stations are received and validated, and the latest information available is kept in memory.

The workstation's purpose is to connect an operator with the system; it consists of a regular keyboard supplemented by function keys. All operator requests are decoded and validated by the workstation. Through the workstation, an operator can get access to alarms in the system, as well as the status of any remote station. Both static and dynamic statistics are available on demand, both of which can be stored for several weeks. A manual override is available allowing decisions about the state of a remote station to be set by the operator.

The database contains information about each remote station, for example the name and ID number, whether the station is on-line or remote, and any alarms or status codes attached to the station. The subsystem in charge of the database allows certain measurements to be stored in timed intervals, (for example, every 15 minutes). The data in memory is synchronized with the data on disk in case of an emergency such as a power failure. It is possible to get a hardcopy of a screen or section of the database, and all information that is received from the remote stations are printed chronologically. There is also a way for a supervisor to be allowed to change any object in the database including the actual names of the stations.

The staff involved in this project consisted of two system specialists from the company, and one engineer from the client. One of the main purposes of the project was to give the client's engineer a chance to familiarize himself with the ABL methodology.

The original functional specifications were translated into the ABL methodology. All specifications, documentation and the subsequent code were written using the ABL methodology. The client found that the system was clear, consistent and had precise documentation at every level; so precise, in fact, that by applying the methodology, errors in the original functional specifications, (due to being vague and unclear), were discovered and modified.

As is often the case, it was necessary to perform some modifications on some of the subsystems. The methodology proved to be invaluable for two reasons. To begin with, because of the methodology, it was easy to understand the logic behind the subsystem, so the changes were made in a well defined manner. This is, of course, not unique to the methodology, but would be the result given any well defined, well documented system. However, the second reason was more unique to the methodology. Each line of code in a programme developed using the ABL methodology has a corresponding line of text. It is this text which forms the detailed design level, and when the design is changed, the code gets changed also. Conversely, when a programme gets changed, the changes are instantly reflected in the documentation. This not only saves time, (the project was finished on schedule), but results in a programme which always matches the documentation, which is almost never the case in other programming environments.

CHAPTER 4

THE USER-INTERFACES OF THE ABL TOOLS

There are two main versions of ABL tools that have been used to design and implement software systems in industry. The first version is called SAM I (the Oracle implementation), and the second version is called SAM II (the dBASE III plus implementation). The ABL methodology is extremely useful for designing and implementing software systems in industry, but there are major shortcomings in the existing ABL tools, which were noticed by the author of this thesis (as well as other system designers) while participating in the aforementioned projects.

To correct the problems encountered with the existing ABL implementations required an entirely different approach. The result is a prototype for a new set of ABL tools which allows the user to work with a syntax-driven wordprocessor instead of using data-entry forms.

In order to understand the necessity for a new set of ABL tools, the rest of the thesis concerns itself with discussing the problems with the existing tools, and showing how the prototype for the new ABL tools resolves these problems.

4.1. THE ORACLE IMPLEMENTATION

It is almost mandatory for a user of the Oracle version of the ABL tools to have a working knowledge of the Oracle database management system because the Oracle version is so primitive. Aside from the problems with the data-entry screens, and the ABL operations that have to be performed manually, just starting the system requires several Oracle commands. These commands have not even been placed in a batch file, which would have at least hidden the details (various switches and qualifiers) of the Oracle command lines from the user.

Once inside the ABL editor things are not much better. In order to design a process, the user is obliged to enter column after column of ID numbers. For example, the alternative screen consists of several sections, where one section is for the alternative's guard, and another section is for the alternative's action flow. The guard table is made up of two fields, an ID field for a condition, and a condition value field which denotes whether in this instance, the condition should be considered true or false. The action flow table is also made up of two fields, where one field is an ID field for an action, and the other field is the position of the action in the action flow. To be able to fill in these tables, the user must have a list of actions and conditions with the proper ID numbers. It is almost impossible to do this without a copy of the ABL objects and their corresponding ID numbers.

It was necessary for the user to periodically drop all the indices and recreate them again. For this task, the user had to have a knowledge of Oracle, as well as an understanding of the relational model of the ABL methodology, as represented by the Oracle tables and indices. There also existed tasks that the user had to write queries for because there was no support for them in the ABL system, for instance, the ability to renumber either cluster or alternative IDs.

What became common practice amongst users of the Oracle tools, was to use an Oracle utility that would allow the database tables to be exported to ASCII files, use a wordprocessor to manipulate the data, and then use an Oracle utility to reload the database from the updated ASCII files, overwriting the database in the process. This "back door" approach was used for two main reasons, the first of which was speed. Oracle was very slow, and the system was made slower because the forms were so cumbersome to use. By contrast, a wordprocessor was able to keep up with the user, and the export and import utilities could be performed in batch mode. The second reason that the "back door" approach was used was because the Oracle system was not really useful, other than for its report generation abilities. Accessing the tables directly was not very different from using the special ABL forms, and it could be done faster, which is certainly important to people working under the pressure of having to produce a system by a certain deadline.

These practices employed by users of the Oracle system contravene the spirit of the ABL methodology. It was felt that for a user to load the database into ASCII files, and then reload the database after manually manipulating the data in the tables, was undermining the use of a database, by treating the information stored in it as simple text.

With the Oracle version of the ABL tools it was almost a necessity to understand the details of the database structure, and to have a working knowledge of the Oracle database management system. This was not as important with the dBASE III PLUS version of the ABL tools, but it was still difficult to use them to properly design a software system.

4.2. THE DBASE III PLUS IMPLEMENTATION

The user-interface of the dBASE III PLUS version of the ABL tools is a hierarchical menu system, which contains two types of displays. One type of display is a read-only menu, and the other type of display is a data entry form. At the bottom of both types of displays are a list of options available for the current display, and the user can select an option by typing its first letter. However, it is not always clear what functions will be performed by choosing an option. In the process menu, for instance, a user is given the option of an edit function or a modify function. Though both options sound similar they allow the user to perform very different tasks. The modify option allows the user to make changes to the various fields of the form which contains the process name, while the edit option allows the user to navigate to the various components that make up a process, (clusters, alternatives, etc).

This example helps to underline one of the weaknesses of the tools - its commands have their roots in the dBASE III PLUS management system. In dBASE III PLUS, the edit command is used to update information in a database table, wnile the modify command is used to change the structure of a database table. Another more obvious example of how the tools reflect the environment in which they were implemented has to do with record numbers.

Every record in dBASE III PLUS has a corresponding ID number. This ID number has absolutely no bearing on the ABL methodology, it only pertains to database storage. Yet throughout the ABL editor, there are places where it is possible to access an ABL object, such as a cluster, by using the record ID. This is in spite of the fact that the record ID has no relation to the cluster, which only serves to confuse the issue for a user, particularly since sometimes the cluster ID matches the record ID number, and sometimes it does not.

Navigating from display to display in the dBASE III PLUS tools is a problem both because the navigation commands are clumsy to use and slow in their response, and also because the navigation commands make it difficult for the user to conceptualize the ABL process that is being designed.

Consider the example of a user who has entered the strategy, (the clusters and alternatives) of a process during a previous session, and is now ready to add some actions to the action flow. Upon entering the ABL editor, the user is confronted with the opening menu. The process menu is chosen next, followed by the cluster menu, which is a split screen display where the top half of the screen contains clusters and the bottom half of the screen contains alternatives. (It should be pointed out here, that even though the information on the display contains the clusters and alternatives for the process, it is not presented to the user in such a way as to resemble a strategy report - this would have been a big step towards allowing the user to conceptualize at least the top-level algorithm, and would not even have been difficult to do. This is a graphic example of how the tools do not reflect even the minimal needs of the user). A cluster is chosen from the cluster menu, and then the user navigates to the bottom half of the display which contains the alternative menu. An alternative is picked and the user then chooses the action flow option. A new split screen is displayed, where this time the top half of the display is the action flow of the alternative, and the bottom half of the display is the action list menu. It should be noted that each menu or submenu which is navigated to requires the system to generate a new screen, and the information that is contained in each screen must be taken from the database. The action flow is added, the user navigates back to the alternative menu, chooses another alternative from the current cluster, (or has to navigate first to the cluster menu in order to choose a new cluster and then back to the alternative menu), and then chooses the action flow option. This is an extremely slow process.

Speed is not the only problem here. It is very difficult to design an ABL process using the dBASE III PLUS implementation. To illustrate this, the process of adding actions to the action flow will be examined. First new actions will be considered, followed by the process of adding existing actions to the action flow. When adding a new action or actions to the action flow, the action flow menu disappears and is replaced by an action form. The form is filled out, and the user has the option of either adding more new actions to the action flow or of going back to the action flow menu. These actions are added in isolation. The user is filling out a form that resembles a form found in a database mailing system. There is no connection with the ABL process, or with the alternative the action flow belongs to. In fact, the user cannot even see the action or actions in the action flow until he is finished adding the action/s.

The second scenario involves adding actions to the action flow that already exist in the environment that belongs to the process. The user navigates to the action list which is the bottom half of the display. This display is read-only. The user can examine several actions at a time, and is allowed to choose the current action. Existing actions are added to the action flow either by repeatedly navigating to the action list display, making an action current, navigating back to the action flow part of the screen, and adding the current action in the desired place, or be forced to use the ID numbers of the actions. The action flow menu has an option that allows the user to enter a string of action ID numbers which results in the referenced actions being added to the current action flow.

Because of user complaints, a second method was introduced which allowed for much faster navigation through the system. Most displays were given a new option called the 'where' option. What 'where' allows the user to do is to go directly to a menu several levels either above or below the current menu, without having to traverse any intermediate menus. So, if the current display is the cluster menu, then the user can immediately navigate to the action flow menu corresponding to any valid alternative in the ABL process. While the where option certainly makes it easier and faster to enter in action flows and other ABL objects, there is a price that must be paid for these benefits. The 'where' option sabotages the ability that the ABL tools had to impose a hierarchical structure on designing an ABL process.

Another way that the tools work to defeat themselves is in the way the system allows an ABL process to be built. Using the process menu as a starting point, the clusters and alternatives (the top-level strategy), should be entered, and then the guards should be added for each cluster, followed by the action flow for each alternative. However, in addition to this method of using the process menu (along with its submenus), the system provides its own "back door". It is possible to manipulate the components of a strategy and an environment directly via the strategy and environment menus (and their submenus). The implications of this, is that a user tends to design an entire ABL process on paper, break it into its components, and enter it into the ABL tools using the strategy and environment menus. The submenus of the process menu are then used to quickly link the strategy and environment by using the ID numbers of the objects that have already been entered. This is hardly promoting the idea of designing software systems using a computer.

References have already been made to the data entry forms that are used to enter ABL objects into the ABL process. Most of these references have been with respect to the inappropriateness of designing something as dynamic as an ABL process using such a static method. However, even if these objections were put aside, there are other problems with the data entry forms that make them untenable.

To illustrate this point, it is only necessary to take the example of the code field for an action, (though this example applies equally to other ABL objects which have code fields as well as other fields in the forms which have similar problems). The first objection to this field is that it requires the user to enter the code for an action into a rectangle. This means that as the user types the code, it will get automatically wrapped around, very often cutting variable names, which makes it hard to read the code corresponding to an action. The second objection is that if the user requires that the code for an action occupy two or more lines, then the user places a backslash (\) where he would like a linefeed to occur. It is only when a report is generated that the backslash gets replaced with a linefeed. This makes it very awkward to examine the code.

It could be argued that these objections simply have to do with aesthetics, and are therefore not important. Leaving aside the issue of whether or not these objections are simply aesthetic considerations, (and also avoiding a discussion on the importance of aesthetics in either case), there is still a third objection, which on its own, raises serious questions about the validity of the entry form approach. This objection deals with the restriction imposed on the size of the code field. In almost every project (and perhaps even every project), where the ABL methodology was used, at some point the code field was found to be too small. The only solution available to the user is to shorten the code, which is not always easy to do. Usually what this involved was putting extra constraints on the length of names given to function calls and variables. To have to work with these kinds of constraints when implementing a software system simply because the tools that are being used do not reflect the needs of the user is unacceptable. Instead of using data entry forms, the tools need some method of representing ABL objects that do not restrict the user, otherwise the tools will only have limited value.

Unlike the Oracle version of the ABL methodology, the dBASE III PLUS version allows the user to renumber the alternatives and clusters of a process without having to make the user write separate stand-alone functions to perform this task. However, this function is not automatic. It is necessary for the user to navigate to the appropriate menu, and pick the renumbering option. The renumbering is performed directly on the database, and the user must wait while this is being done.

Aside from problems with navigation and data entry, the dBASE III PLUS version of the ABL tools has other drawbacks. One of these drawbacks is the restrictions that are placed on the user by the tools. For the most part, these restrictions have been arbitrarily decided by the person implementing the tools, and are not a consequence of the ABL methodology, the user's needs, or the tools themselves.

It is obvious that any computer system will have some restrictions. ID numbers, for example, cannot be infinite, at the very least, they depend on the size of the underlying data object used to represent them in a computer. There is also some validity in imposing some restraints on the size of ABL objects. The dBASE III PLUS implementation has a limit of thirty-six alternatives for a cluster and ninety-nine alternatives for a strategy. Though these numbers appear to be arbitrary, it could be argued that ABL processes tend to be too unmanageable and hard to understand as they get very large, and so, certain constraints are justified. However, certain restrictions that are placed on the user by the ABL tools serve no purpose and actually hinder the user on occasion. A good example of this is the previously mentioned problem with the code field of data entry forms which restricts the size of a line of code.

Another example of a restriction that the ABL tools impose on the user is that the dBASE III PLUS tools only allow ten conditions to appear in a cluster's guards. Though this might have been done because it was felt that allowing more than ten conditions would result in unwieldy guards, nevertheless, there are times when this limit is not adequate. The most common example of this is a function which has the task of monitoring a keyboard. This function will probably require guards which contain well over ten conditions, and this is a case where the addition of extra conditions should not interfere with the ability of the user to understand the logic of the ABL process. The solution in this case, would be for the user to arbitrarily break up a cluster into two or more clusters. This actually makes the ABL process harder to understand because the reports do not show any reason for having several clusters. In fact, it is only done to accommodate the tools.

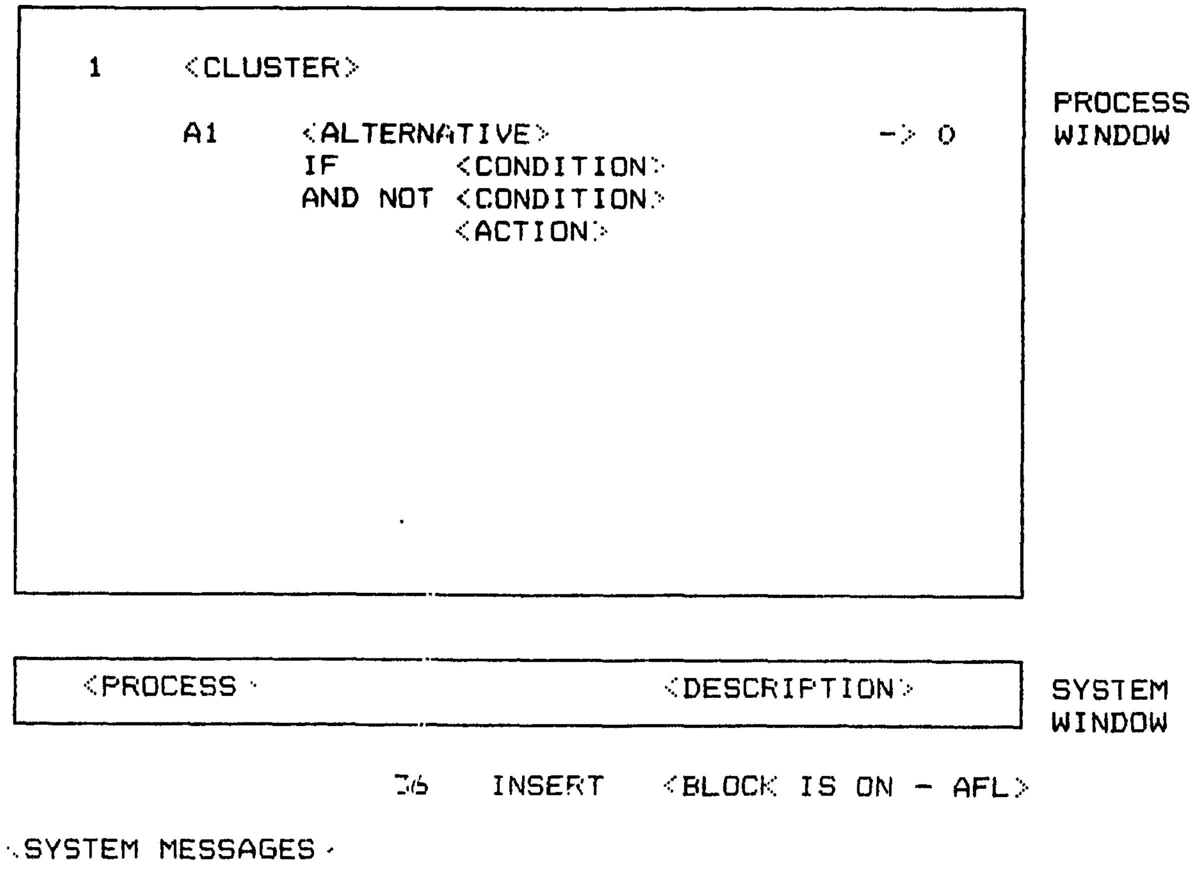
Perhaps the most serious drawback of this version of the ABL tools is that of side-effects. These are changes that the system implicitly makes to an ABL process which are done without the user's knowledge. A prime example of a side-effect occurs when a user wishes to make a copy of an ABL process. Many users, (both experienced users and novices), have tried to make a copy of an existing ABL process by making copies of its components (environment and strategy), only to find that the new ABL process is different from the original one. This is because when the dBASE III PLUS version of the ABL tools copies an environment or a strategy, it will very often change the ID numbers of the ABL objects contained in these components.

The rationale behind this decision to modify these ID numbers, is that since the ID numbers have no bearing on the ABL methodology, it should make no difference if the system decides to change them. There are two major flaws with this line of thinking. For one thing, there is something wrong with a system that requires a user to manipulate objects using their ID numbers, then changes these ID numbers without being instructed to, and without even notifying the user, on the basis that the ID numbers are not important. Even more important, is that this side-effect creates a conflict between the ABL tools and the ABL methodology. Given that a process is composed of a strategy and an environment, two processes with the same strategy and environment should be functionally equivalent. Yet because the system potentially changes a strategy and an environment when a duplicate is made of them, this no longer holds.

4.3. THE NEW ABL TOOLS

In order to address the various problems in the existing ABL implementations, in particular with respect to the user-interfaces, a prototype for a new set of ABL tools has been implemented. The following sections describe the user-interface of the prototype.

4.3.1. Process Window



```
1     <CLUSTER>

      A1     <ALTERNATIVE>                       -> 0
             IF       <CONDITION>
             AND NOT  <CONDITION>
                      <ACTION>
```

```
   <PROCESS>                               <DESCRIPTION>
```

```
                       36    INSERT    <BLOCK IS ON - AFL>

<SYSTEM MESSAGES>
```

figure 4.1

The user interface of the prototype for the new ABL tools consists of a standard screen twenty-four lines long, and eighty columns wide (see figure 4.1). The first twenty-one lines of the screen is the process window. It contains the ABL process that is being edited by the user. The remaining three lines make up the system window, which is used by the system to communicate with the user.

The process window has four types of lines, a cluster line, an alternative line, a condition line and an action line. The fields of each line type are described from left to right as follows. The cluster line has two fields, the cluster ID field, and the cluster description (or narration) field. The alternative line has three fields, the alternative ID field, the alternative description field, and the next cluster ID field, which is preceded by an arrow. The condition line has three fields. The first field is the condition prefix. For the first condition in a guard, the condition prefix will be "IF", and for subsequent conditions in a guard the condition prefix will be "AND". The other fields belonging to the condition line are the condition value field, which is either blank or "NOT", and the condition description field. The action line just contains an action description field.

With the exception of the next cluster ID number, the ID number fields are not accessible to the user. Both the cluster IDs and the alternative IDs start at ID number one, and continue in ascending order until there are no more clusters or alternatives. If a cluster or an alternative is added or deleted, all the relevant ID numbers are updated automatically.

There are only three field types that the user has access to,

the description fields (for all line types)
the next cluster ID field in the alternative line
the condition value field in the condition line

Normally the cursor stays on the description field of whatever line it is on. In order to access the next cluster ID field or the condition value field, it is necessary for the user to press the shift-tab keys. If the cursor is on a condition line and the user presses the shift-tab keys, then the condition value field is toggled between "NOT" and blank, (where blank signifies true and "NOT" signifies false). If the cursor is on an alternative line and the user presses the shift-tab keys, then the user can change the value in the next cluster ID field.

The first thing that happens when a user presses the shift-tab keys while on an alternative line, is that the next cluster ID field changes to reverse video. By pressing the down arrow key or the up arrow key the user can increase or decrease the value of the next cluster ID. The range of the next cluster ID is between 0, which designates exit process, and a value one greater than the largest cluster ID number. When the user is finished choosing a new next cluster ID, pressing the shift-tab keys changes the next cluster ID field back to regular video, and allows t ie user to resume editing from the alternative description field. If the new next cluster ID has a value one greater than the largest existing cluster ID number then the system automatically creates a new cluster at the end of the ABL process (with alternative).

One of the by-products of changing the next cluster ID number, is that the cluster that was referenced by the original next cluster ID number might become orphaned; it is possible that there is no longer a path leading to that cluster. If this is the case, then the system will warn the user that a cluster has been orphaned.

It could be argued, that for consistency sake, when a cluster has been orphaned it should be deleted automatically, given that when a next cluster ID references a new cluster it is automatically created. However, from the point of view of the user, it is better to have the orphaned cluster deleted explicitly.

4.3.2. System Window

Line twenty-two is the first line belonging to the system. It is in reverse video and contains two fields. On the left hand side of the line is displayed the name of the process that is being edited, and on the right hand side of the line, a process description, which can be up to sixty characters in length, is displayed.

The twenty-third line of the screen has three fields. The first field is the relative column position, the second field is the insert key indicator and the third field is the block buffer status field.

The column position is referred to as the relative column position because it does not give the user the column position from the beginning of the line on the screen, but rather, the position from the beginning of the current ABL object. Though each ABL object starts at a different column, the start of each ABL object is designated as column one by the relative column field.

The insert key indicator will either display the word "Replace" or the word "Insert". When the insert key indicator is set to Replace, any character that the user enters will overwrite the character at the cursor location. If the insert key indicator is set to Insert, the character at the cursor location will not be overwritten, instead it will make room for the new character. The cursor will be represented by an underscore character in Replace mode, and a rectangle character in Insert mode. By pressing the insert key, the insert key indicator is toggled between the two values. The default value is Replace.

The block buffer status field indicates whether or not there is any object in the block buffer. If the buffer is empty then the block buffer status field is blank. When a block is created, the message "BLOCK IS ON - XXX" is displayed in bold letters, where XXX will be CLS, ALT, GRD, AFL, CND, or ACT, which denote cluster, alternative, guard, action flow, condition, or action, respectively.

The last line of the display is reserved for system messages to the user. These messages are displayed on the left-hand side of the line in bold letters, and are sometimes accompanied by a beep. On occasion, the system requires the user to press a key after reading the message, in order to continue, or to enter a yes or no answer in response to a question. Some examples of system messages are,

"Undefined key"

"Are you sure? Type y or n"

"Loading strategy into memory".

4.3.3. Navigation

Like all full-screen editors, the ABL editor allows the user to navigate both horizontally and vertically. Horizontal navigation moves the cursor without leaving the current line, and vertical navigation moves the cursor to a different line, sometimes causing the display to change in the process.

It is possible to move to the beginning or to the end of the line that the cursor is currently occupying. A line can be traversed eight characters at a time by pressing the tab key, or the cursor can be moved one character to the left or one character to the right by pressing the left arrow key or the right arrow key. Note that the shift-tab keys can be viewed as horizontal navigation keys which allow the user to navigate to fields that could not otherwise be accessed.

It is possible to move to the top or to the bottom of the ABL process. If the top (or bottom) line is on the current screen then the cursor is moved to it, otherwise the first screen (or last screen) is displayed, and then the cursor is moved to the top (or bottom) line. It is possible to move the cursor up a line or down a line with the up arrow key or the down arrow key. It is also possible to page through the process, forwards or backwards, one screen at a time, by pressing the PageUp key or the PageDn key.

4.3.4. Editing

There are two types of editing that is performed by the ABL editor. There is editing that is performed by adding and deleting characters on the description field of the current line the cursor is on, which can be considered horizontal editing. There is also editing that effects the size of the ABL process, by adding and deleting ABL objects, which can be considered vertical editing.

Of the two types of editing, the horizontal editing is much more straightforward. If the key that the user presses is not a command key then it is taken to be data, and is added to the current description field (with the insert key status taken into consideration), provided it is not full. If the DEL key or the backspace key is pressed then a character is deleted, provided there is a character to delete. It is also possible to erase the contents of the current description field entirely, or to erase the contents from the cursor to the end of the description field.

Things become more complicated when we enter the realm of vertical editing. Unlike a normal wordprocessor, where every line is considered a separate entity, the ABL editor deals with ABL objects, some of which are complex objects. When a cluster is added or deleted from a process, the alternatives that belong to it are also added or deleted, and in addition, the guards and action flows which belong to the cluster's alternatives also get added or deleted. Furthermore, there are certain restrictions as to how an ABL object can be deleted, and also where a new ABL object can be added.

Up until now, most of the functions that the ABL editor performs automatically, are functions that are designed to help make the task of creating ABL processes easier for the user. For example, when a next cluster ID number is assigned a cluster ID number which does not exist, the editor will create the cluster automatically, thereby relieving the user of the necessity of creating the cluster later on. However, not only must the ABL editor serve the user, it must also be responsible for making sure that it remains faithful to the rules of the ABL methodology. The

ABL editor cannot allow a user to insert an action flow in the middle of an alternative's guard, for instance.

To allow the user to do something at the expense of the ABL methodology would be an error of the same magnitude as was made in the previous versions, where it was felt that the ABL methodology was first and foremost, even at the expense of the user. This issue is being addressed now, because it is with vertical editing that the editor becomes an ABL editor, rather than a glorified data-entry system.

4.3.5. Adding an ABL Object

There are two ways to add an ABL object in the ABL editor (not including copying a block). The first method is more natural for the user, but it cannot be used all the time. The second method is more complex, but there are times when it is necessary to use it.

Quite often when creating an ABL process, the next line to be added is of the same type as the current line that the cursor is on. For example, when working on a strategy report, there are almost always several alternatives for each cluster, and when working on a process report, alternatives have several conditions in their guards, and several actions in their action flows.

When the user presses the return key, the ABL editor will create a new object below the current line that the cursor is on, of the same type as the current line. If the cursor happens to be on the cluster with ID number one, which is always at the top of a process, then the user is given the choice of either creating a new cluster below cluster one, or having the new cluster itself become cluster one.

One of the basic rules of the ABL methodology, is that every cluster must have at least one alternative associated with it. To this end, when a cluster is created, an alternative is automatically created also, with a next cluster ID of 0.

When a new ABL object is created, the description field is automatically initialized with the name of the object type in angle brackets. For example, if the object being added was an alternative then the description field would be <alternative>. This "place holder" is useful to the user, for without it blank description fields could become confusing. This is particularly true of actions because there is no other field in an action line.

Though the addition of ABL objects through the use of the return key is very natural to the user, it is not sufficient. If it were the only method for adding objects, then it would only be possible to have a strategy report, and never a process report because there would be no way to add a first condition or a first action to the process. In order to add an ABL object that is different from the object type of the line the cursor is currently on, the user presses the F9 key.

Pressing the F9 key causes a window to open on the screen, with four choices, cluster, alternative, condition, action, labeled one to four. The user can either move the cursor, (which is the size of the choices), to the desired choice by using the left or right arrow keys, and then press the return key, or can simply enter the number which is beside the desired choice. If the user decides to abort the operation, pressing the escape key will close the window. Otherwise the editor will create the new ABL object below the current line, providing that the rules of the ABL methodology are observed.

In order to add a cluster to the ABL process the following rules have to be observed. To begin with, a cluster cannot be added after a line which is of type cluster. If this were allowed, then the existing cluster would no longer have an alternative, because directly after the cluster would be another cluster. Likewise, in order to add a cluster after a line of type alternative, the alternative cannot have any conditions or actions. Otherwise the conditions and actions would end up cut off from the alternative they belonged to. The other restrictions to adding a cluster apply to conditions and actions. For a cluster to be added after a condition, the condition has to be the last condition in the guard, and there must be no action flow for the alternative. Finally, for the cluster to be added after an action, the action must be the final action of the action flow.

If the user wanted to add an alternative, there would be no problem if the current line type was a cluster. There is nothing wrong with adding an alternative between a cluster and its first alternative. If the current line type was an alternative, however, there could not exist any guard or action flow for the current alternative. Just as in adding a cluster, if the current line type is a condition then there can be no more conditions, and no action flow, and if the current line type is action, then the action must be the final action of the action flow. All alternatives are created with the next cluster ID set to 0.

There are only two places that a condition can be added. The type of the current line has to either be a condition, in which case the new condition is added to the guard, or the current line has to be of type alternative, in which case the condition becomes the first, and possibly only element of the alternative's guard. The condition value field of the line containing the condition is initially blank (new conditions are originally set to true).

The final ABL object that can be added is an action. (While it is possible to block and copy a guard or an action flow, it is not possible to create an empty guard or an empty action flow). The legal types for the current line when adding an action are alternative, condition, or action. If the current line is an alternative, then the alternative cannot have a guard, because an alternative cannot have an action followed by several conditions. Likewise, an alternative's guard cannot have an action imbedded in it, so if the current line type is condition it must be the last condition in the guard. There are no restrictions when the current line type is an action.

If the ABL object that has been added is either a cluster or an alternative, then it is quite possible that some ID fields will have to be renumbered. In the case of an alternative, renumbering alternative IDs is quite simple. If the new alternative is not the last alternative in the ABL process, then all remaining alternatives will have their

ID numbers incremented by one. However, in the case of adding a cluster, there is a little more work to do. Aside from adjusting the cluster ID numbers, there is also the matter of the next cluster IDs. They must also be adjusted, or else they will no longer provide the path that the user originally intended. Furthermore, if the block buffer is not empty, and the block contains any alternatives, then the next cluster ID of each alternative in the block must also be examined, and adjusted where necessary. All ID renumbering is done automatically by the ABL editor, and is transparent to the user.

4.3.6. Block Commands

An ABL object can be also be added to a process by appending the contents of the block buffer after the current cursor position. There are three block commands that are available to the user, make a block (alt-b), copy a block (alt-c), and undo a block (alt-u). The rules which govern where the contents of the block buffer can be added in a process are identical to the rules which have been outlined in the preceding section on adding an object using the F9 key.

The block commands in the ABL editor functions in a similar manner to cut and paste commands found in many ordinary editors, except that, normally when a block of text is cut, thereby initializing the block buffer, the block is removed from the original text, but in the ABL editor, the original text remains unchanged.

When the user presses the make block keys, the block buffer is initialized with the ABL object contained in the line that the cursor is presently on, and the block buffer status field displays the message "BLOCK IS ON - xxx", where xxx is a three letter mnemonic representing one of the ABL objects (CLS, ALT, GRD, AFL, CND, and ACT). If the buffer block has already been initialized, it will be necessary for the user to uninitialize the buffer block by pressing the undo block keys.

If the block buffer is being initialized with a cluster or an alternative, then the buffer will contain the complete ABL object. In the case of a cluster, all of its alternatives (along with guards and action flows), will be contained in the buffer, and likewise, in the case of an alternative, the guard and action flow (if any), will also be in the block buffer. When the block is copied to a process, the ID numbers for the clusters alternatives, and next clusters will be replaced according to where in the process the buffer is being added.

If the user presses the make block keys, and the line that the cursor is currently on is an action or a condition, the system will display a prompt which will allow the user to choose between initializing the block buffer with the simple ABL object (the action or the condition), or the complex ABL object (the action flow or the guard).

So far, adding an ABL object to a process using the block commands, seem to be identical to adding an ABL object using the F9 key, with the sole difference being that the block commands are the only way for the user to be allowed to add an action flow or a guard. However, there is a very important distinction to be made between the two methods when adding an action, or a condition, (and an action flow, or a guard), to a process.

Consider the example of a cluster which has two alternatives. One of the alternatives will be chosen in the event that the variable status is equal to the value of the constant SUCCESS, while the other alternative will be chosen if status does not have the value of the constant SUCCESS. In this example, it would not be appropriate to use the F9 command for creating the two guards because this would lead to two copies of the condition "status is successful". What would be preferable, would be to have one copy of the condition "status is successful" with one alternative's guard referencing the condition as true, and the other alternative's guard referencing the same condition, only this time, with the value false.

This is what happens when the block commands are used with actions or conditions (or action flows or guards). The block will not create a new condition or action. It will only create a member of a guard or action flow that will reference the action or condition that was originally blocked using the block command. Therefore, when taking a strategy report, (which just describes the top level algorithm of a process), and extending it into a process report (which describes the algorithm of the process in detail, complete with actions and conditions), the method that is the most consistent with the ABL methodology would be to create the guard for the first alternative in the cluster, block the guard using the block command, then copy the guard, (using the block copy command), to the other alternatives in the cluster, changing the condition value fields to ensure mutual exclusiveness between the guards.

While this technique is not as important with respect to actions, it is still in keeping with the philosophy of the ABL methodology to not have identical actions in a process. Instead, there should only be one copy of an action which can be referenced several times by an action flow or by different action flows. It is only with use of the block commands that this can be done.

4.3.7. Deleting an ABL Object

In a regular wordprocessor deleting a line of text is done by invoking the delete command while the cursor is on the line of text to be deleted. The wordprocessor removes the deleted line from the display, and places the cursor either on the line above or below where the deleted line used to be.

Deleting an occurrence of an action or a condition from an ABL process is done in much the same way. The use of the word occurrence is significant, because the action or condition might be used in several places. It is only when the last reference to an action or condition is deleted that the action or condition itself is removed from the process. A field will eventually be added to lines containing actions and conditions which show the user how many times the current action or condition is referenced in the process.

Aside from actions and conditions, deleting ABL objects is not as simple as deleting a line of text from a wordprocessor. There are two major differences from regular text. The first difference is that since most ABL objects are complex, we are dealing with an object that potentially spans several lines, and so the ABL editor has to be able to determine the extent of the object to be deleted. The second difference, is that with a regular wordprocessor, the line of text that the user requests to be deleted can always be deleted. In the ABL methodology certain rules must be followed, and the ABL editor must make sure that the user request is legal. For example, as stated earlier, one of the fundamental rules of the ABL methodology, is that every cluster must contain at least one alternative. Consequently, if a user makes a request to delete an alternative, and the alternative is the only alternative in the cluster, then the ABL editor, rather than deleting the alternative, must display a system message telling the user that the only alternative of a cluster cannot be deleted.

Before a user request to delete a cluster can be carried out, certain checks have to be performed by the ABL editor. For one thing, the minimum requirements for an ABL process is to have one cluster which in turn has one alternative. The guard and action flow of the alternative can be empty, and the next cluster ID can have a value of 0. (If code were generated for this process then the programme would be the equivalent of an exit statement). If such a process existed, and the user requested that the cluster be deleted, the ABL editor would not allow it.

In the ABL methodology, whenever an alternative is chosen, its next cluster ID will determine the path that the programme will take. If the user requests the system to delete a cluster which is referenced by an alternative's next cluster ID, (where the alternative does not belong to the cluster to be deleted), then the editor would not allow the cluster to be deleted. As an extension of this, if the cluster to be deleted is referenced by an alternative's next cluster ID in the block buffer, then the editor would also not be allowed to delete the cluster.

The reason for this decision lies in considering the other options available to the system. One possible solution is to keep the alternative's next cluster ID number as it is and delete the cluster. The problem with this is that we now have an alternative whose next cluster ID does not lead to the cluster that the user intended it to, and the user is not even aware of the change. Furthermore, if the cluster that was deleted was the last cluster in the process, then the next cluster ID does not even point to a cluster ID that exists.

Another option available to the system would be to change the next cluster ID and then delete the cluster. Two possibilities immediately come to mind. Either set the next cluster ID to 0, or have a reserved ID (such as -999) which would signify that the next cluster ID is no longer valid. However, what is happening here is that the system, unknown to the user, is making changes to the process. It is important for this kind of side-effect to be avoided. The user should always be in control of changes to the control flow of the process. Also, by forcing the user to change the next cluster ID, manually prior to deleting the cluster, makes the user reflect on the implications of his actions. If the alternative will no longer lead to the cluster that is being deleted then the user will have to decide on the new path. Perhaps the alternative will no longer be needed and the user can delete it, or maybe after the user has examined the process from the point of view of the alternative, whose next cluster ID references the cluster slated to be deleted, the user will realize that the cluster should not be deleted. Deleting a cluster that is actually necessary to the process can cause a lot of inconvenience to the user because it can be quite large, what with alternatives and guards and action flows. In the final analysis, its best to only delete clusters that have been orphaned.

Once the system has established that the cluster is an orphan, and that there exists at least one other cluster in the process, then the cluster will be deleted, along with all of its alternatives. The cluster IDs, the alternative IDs and the next cluster IDs, (in the block buffer also), are all adjusted accordingly. The display is updated or completely redrawn, and the user is warned that a cluster may have become orphaned, (because the deletion of a next cluster ID belonging to one of the alternatives that was deleted along with the cluster may have been the only path to a cluster).

The only case in which an alternative cannot be deleted is when it is the only alternative belonging to a cluster. Otherwise an alternative can always be deleted. When an alternative is deleted, the guard and action flow associated with it, if any, are also deleted. The remaining alternative ID numbers are updated automatically.

When an alternative is deleted, the system checks to see whether the next cluster ID of the alternative was the only path to the cluster it references. If so, the system displays a message warning the user that a cluster has been orphaned. It is necessary to allow the user to orphan clusters, because in order to delete a cluster, it must be an orphan.

There is no facility for deleting either guards or action flows. Normally these will not be very large, and the user can delete them by deleting their individual components, (conditions and actions).

4.3.8. Report Generation

In the Oracle version of the ABL tools, the user generates a report from the DOS prompt. The dBASE III PLUS version has the report generator integrated into the ABL system, but it is necessary for the user to navigate to the report menu in order to generate a report. In contrast to the existing implementations of the ABL methodology, the new tools have the report generator included in the ABL editor. Report generation in the new tools is akin to printing a file from a regular wordprocessor. The user presses the report generator key (F7), the system opens a window in the display which prompts the user for a report type, and a report is generated.

There are certain report options which the user will be able to set by pressing the configuration key (F10). For example, it will be possible for the user to choose whether the reports are to be sent to a file or to the printer. If the file option is chosen, it will be possible for the user to rename the destination files, as well as choosing the disk that the file will be written to. Whatever the destination of the reports, it will be possible for the user

key	Description
INSERT	Toggle between insert and replace mode
right arrow	move cursor one to the right
left arrow	move cursor one to the left
up arrow	move cursor up one row
down arrow	move cursor down one row
DEL	delete character cursor is on
CTRL-backspace	delete current line
backspace	delete character left of cursor
RETURN	add new object (same type as current)
PGUP	display previous screen
PGDN	display next screen
HOME	move cursor to beginning of line
END	move cursor to end of line
CTRL-HOME	move cursor to op of process
CTRL-END	move cursor to end of process
TAB	move cursor 8 spaces right
SHFT-TAB	toggle NOT or set Next Cluster
F4	quit without saving
F5	clear line (line still exists)
F6	clear line from cursor
F7	generate report
F9	add new object (specify type)

figure 4.2

to choose from among several page sizes.

The prototype is only capable of generating two report types. One report is the strategy report, a report which consists of just cluster and alternative descriptions (along with the next cluster IDs), and the other report is the process report, which has, in addition to the clusters and alternatives, includes the guards and action flows of a process as well. Sample reports have been included in the appendix. There are several types of reports that the system will be capable of generating, and more types can be defined to suit the individual user's needs.

A summary of the functions and the corresponding keys for the ABL syntax-driven editor is given in figure 4.2.

CHAPTER 5

THE INTERNAL STRUCTURE OF THE ABL TOOLS

5.1. THE ORACLE IMPLEMENTATION

The Oracle implementation will not be considered here in any detail, because it does not really have a clearly designed internal structure. The user enters information via forms created by the Oracle form generator, and generates reports via the Oracle report generator. These generators are manually invoked by the user. The Oracle implementation was not designed with any attempt at integrating its various components.

5.2. THE DBASE III PLUS IMPLEMENTATION

In contrast to the Oracle implementation, the dBASE III PLUS implementation presents the user with a more integrated system. But the internal structure of the tools is weak, and this is partly to blame for a user-interface that is unacceptable, (which is discussed in the section on the user-interface). There are two main factors that are responsible for the weak internal structure, the dBASE III PLUS computer language that is used, and the overall design strategy.

The dBASE III PLUS programming language is not really suitable for implementing a project concerned with creating an environment which allows the user to develop and maintain software. Its prime capability, not surprisingly, is to allow people to have access to the database. The language makes it easy to create tailor-made screens, to get information from the user, and to read and update the database. It was designed, not so much as a programming language, but rather, as a means of automating database queries that were frequently needed by the user. It is ideally suited to an application like a mailing system, or an inventory system, or any application that requires the filling out and printing of forms.

The problems involved in trying to use the dBASE III PLUS programming language instead of a regular high-level language become more obvious when we consider the limitations of its data objects and the data structures that are available to the programmer. The first major flaw is that data objects cannot be declared. If you want to use the variable temp, and you want temp to be of type integer, the only way to do this is to assign an integer value to the variable temp. This goes against one of the fundamental principles of good programming practice, that is, that all data objects used by a programme should be explicitly declared.

Another major drawback in the programming language is that it does not support data structures, (other than those pertaining to the database). Even arrays do not exist in this language, which is unfortunate, because normally an array could be used to create more complex data structures, for example, linked lists, or stacks. However, even if arrays did exist, most data structures could not be defined, because the language does not allow the user to create a data type. The only legal data types are the ones that are built into the language. It should be clear that the dBASE III PLUS programming language is not an ideal choice for developing software systems.

While the dBASE III PLUS programming language was certainly a contributing factor to the weakness of the tool's internal structure, it was not the only determining factor. The overall design strategy, or lack thereof, also was at fault. Originally, the dBASE III PLUS version started out as a pilot project to test the feasibility of creating a cheap version of the ABL tools that was more than just a data-entry system (as in the Oracle version). It was well known by the people who used the Oracle version, that it was much too slow and did not even have proper facilities for entering processes.

Given the time constraint, and considering that the tools were a one person project, it must be conceded that the prototype for the dBASE III PLUS version would not have been realized using normal software design techniques. However, because the person working on the tools was a dBASE programmer, and he started with the Oracle database tables, the tools ended up being a dBASE application programme with a hierarchical menu structure. From the point of view of the people who worked with the ABL tools, this was not substantially better than the Oracle version. Unfortunately, development never proceeded past the prototype stage. Instead, the prototype became the new ABL tools, and the person who wrote the dBASE version became the person in charge of maintenance.

5.3. THE NEW ABL TOOLS

The internal structure of the new implementation of the ABL methodology is completely different from any of the preceding implementations. But it is not just the structure that is different, the entire process of designing and implementing the new tools involved a radical departure from previous efforts.

The most noteworthy difference between the new version and its predecessors is that the new version has been completely designed and implemented using the ABL methodology. What better way can there be to show the capabilities of the methodology then to implement the tools supporting it in the methodology itself? Eventually, the tools will be integrated into the ABL database so that future versions of the tools will be modified using the tools, with the final code generated from the database. This is not unlike having a Pascal compiler, that is in itself, written in Pascal.

The specification phase of the new ABL tools started by dividing the system into two parts, the ABL database and the user-interface. By considering the system in parts rather than in its entirety, it was hoped that the previous flaws in the user-interface could be avoided.

The decision to use a database made sense, (see the discussion in the database section),and the design of the ABL database was straightforward owing to the work that had been done in the previous versions.

With the requirements needed to represent the ABL methodology using a relational database clearly defined, the next step was to create the requirements for the user-interface. This is where the new tools would differ drastically from the existing tools. The most natural way to create a programme is with a full screen editor, and therefore, it was necessary to have full screen editing facilities for the ABL tools, because anything less would be awkward to use for designing and implementing software systems.

In the previous ABL tools, when a process was edited by the user the database was updated immediately. However, in order to support full screen syntax-driven editing facilities, this would have to be changed. To keep the process decomposed in the database and have to continually access the disk is not efficient enough for a full screen editor. Instead, at the beginning of a session, the process to be edited must be read from the database tables and loaded into memory structures. The database must only be updated with changes made to the process when the user explicitly requested it, much the way a word processor works.

At this stage, the functionality of the new ABL tools was specified, and the data objects, (the database and the memory structures), had been defined. The next step was to make a detailed design. The system was divided into subsystems according to specific functions that needed to be performed. There were five subsystems in all, as well as a group of general purpose utilities which could be used by all the subsystems. The general purpose utilities contained such functions as an error handling routine. Each subsystem has a three letter mnemonic, and a function belonging to a subsystem would start with the subsystems three letter mnemonic.

5.3.1. Build Subsystem - BLD mnemonic

The Build subsystem has two main functions. The first function involves the database. When a process is to be loaded into the ABL tool, the Build subsystem is given the ID number of the process. Using the process ID number, it is the task of the Build subsystem to gain access to the database, in order to extract the records belonging to the process. If there are I/O problems with the database tables, then the Build subsystem notifies the system so that it can exit gracefully.

The second function of the Build subsystem is to build the structures in memory that will hold the process, and to load the process into these memory structures. It is possible that there will not be sufficient dynamic memory to hold the process, (but tests run on a PC computer show that this is unlikely), in which case the Build subsystem notifies the system so that it can take appropriate action. If the process that was being loaded was the first process in the ABL tool, then it will exit gracefully, otherwise it will advise the user that there are too many processes loaded in memory at one time.

5.3.2. Screen Subsystem - SCR mnemonic

The Screen subsystem, as its name implies, has the function of looking after the user screen. Any data that is to appear on the screen, with the exception of user messages, must go through the Screen subsystem. Any time that an object is added or deleted from the screen, or a PageDn or PageUp is requested, or a vertical arrow key is pressed, or any other navigational command is invoked, the Screen subsystem will make sure that the screen is updated or redrawn, if necessary.

As well as updating the screen, the Screen subsystem has the further responsibility of maintaining certain data objects related to the screen. There is a screen map, which contains the contents of each line that is on the screen, as well as the type of each line, where a line can be either cluster, alternative, condition, or action. It must keep track of the information pertaining to the cursor, the row and column offsets for the cursor, whether the cursor is visible or invisible, whether the cursor is in replace mode (underscore), or insert mode (block). When the cursor is moved, it is responsible for updating the relative column number. It must keep track of the current line that the cursor is on, including the type of line it is on, and what attributes the line possesses (reverse video, bold, normal).

5.3.3. Report Subsystem - RPT mnemonic

The Report subsystem has the function of generating reports. The present version of the tools only allows reports to be generated from the editor, but eventually the report generator will also work in batch mode. Aside from generating reports, it also generates programme code for an ABL process. Though tests have been performed with the code generator, the prototype does not create a programme code report.

The Report subsystem allows the user to choose from certain options when generating the reports. It is possible, for instance, to choose from among several column settings, (80 columns, 96 columns, or 136 columns). As well as changing certain characteristics of the reports, it is also possible to choose their destination.

A report can be sent to either the printer or a file, where the default setting is for a file. The file resides in the default directory, but it is possible to choose another directory, or even another disk drive.

The idea behind these options is to allow the user to customise the development environment. The system is set up in such a way that normally the default settings will be all that is needed. In the future it will be possible to change the default settings.

5.3.4. Modify Subsystem - MDF mnemonic

When an ABL object is added or deleted there are two separate operations that must be performed. The screen must be updated, which is handled by the Screen subsystem, and the memory structures must be modified, which is handled by the Modify subsystem.

The Modify subsystem is in charge of all the pointers, including the block buffer pointer, that point to the various memory structures that contain a process. It is also responsible for keeping track of which object in which structure is the current object that corresponds to the current object on the screen.

5.3.5. Save Subsystem - SAV mnemonic

Given a process ID number, the purpose of the Save subsystem is to find all the information in the memory structures that belong to the process with the process ID number, and to update the database with the information.

As in most word processors, the ABL tools have two options for saving, save and continue, or save and exit. When either of these commands are requested the Save subsystem is invoked. It should be noted that the prototype does not use the Save subsystem.

Both the older versions of the ABL tools and the new version are implemented on an IBM PC compatible computer running under the DOS operating system. However, the programme environment for the new ABL tools is completely different from that of the previous implementations.

The fundamental reason for designing a new version of the ABL tools was that the user-interface did not meet user requirements. For designing and implementing software systems, what is needed is an environment which has a full screen editing facility. In order to create such a facility, it was necessary to be able to use certain data structures. There were data structures needed for the screen, and data structures to hold the ABL process in memory. While the older versions of the tools have been implemented using a database management system, the capabilities of a database management system was just not good enough for building these data structures, and so a high-level programming language was needed.

The new version of the ABL tools has been implemented using the Lattice C compiler version 3.2, (large programme, large data model), with the addition of two libraries. The compiler uses the Lattice dBC III library, a library that allows a programmer to create, modify, and delete dBASE III PLUS files, (database, index, etc.), and the ESI Utility library. The ESI Utility library contains functions for quick screen access, which is essential for a full screen editor.

Even though a database management system was not powerful enough to implement the system, the use of a database was still a high priority for the ABL tools. The choice of high-level programming language had to be one, such that, it had support for gaining access to a database.

Another reason for using the C language instead of a database management system is the portability. The programmes which make up the new ABL tools have adhered to the proposed ANSI standard for the C language whenever possible. There will have to be changes made to the programmes, (in particular, the libraries will have to be replaced), but the programme segments which are not portable have been well defined. Eventually, the new ABL tools will be transferred to other computer systems, such as the Macintosh, workstations, the Novell network, and perhaps even a VAX mini-computer. The current design will have to be modified in the case of multi-user systems.

CHAPTER 6

THE USE OF A DATABASE IN THE ABL TOOLS

6.1. BENEFITS OF USING A DATABASE

In each of the implementations of ABL (both new, and existing [26][32]), the components that make up a process are represented relationally and stored in a database. Keeping a process decomposed in this manner allows multiple views of the process to be extracted. The views, or reports, may be produced at any time and in a variety of formats including a tabular and a graphic format. One of the views is code for the programme.

The programmes that will be produced from the database can be modified to suit the individual tastes of the user without having to edit or change individual programmes, since the programmes are assembled from the database using a report generator. When developing a system, programmes can be generated which produce a trace during execution. What is especially useful about this feature is that the trace that is produced reflects the design logic; by showing what alternatives have been chosen from a cluster, the designer is actually seeing the specification being executed. This has proven to be an invaluable aid in system development [17].

The database allows for static verification and consistency checking as well as dynamic testing and debugging techniques. Every cluster can be checked for completeness and consistency. Because the processes reside in a database the static tests can be performed either in batch or interactive mode.

By allowing the reports to be generated, instead of having to document manually, and by having the lines of code in a programme correspond to lines of description, the documentation remains complete and consistent. This is an extremely rare phenomenon in software development. Normally documentation is produced after the fact, and quite often changes that were made to a system, particularly late in the implementation stage, are either documented in an incomplete manner, or else do not show up at all in the documentation.

Another benefit derived from keeping a process decomposed is that the components are reusable. Quite often a subsystem will contain programmes that operate on the same data objects or perform the same task on different data objects. Rather than starting from scratch, it is possible to make use of the existing components either by making a new copy of the environment (or strategy), or by sharing an environment (or strategy) between two processes.

In previous implementations [26][32], a strategy or an environment could be shared between two or more processes, but in this version, it is only possible to make separate copies. While in theory shared environments or strategies seem like a good idea, in practice certain problems arise.

One problem encountered with shared components is that of side-effects. A programmer makes a change to a process with a shared environment by deleting an action. It is very likely that the action that was deleted was in fact needed by another process, and that now the other process will not work properly. There is no indication that an object is shared, the only information available is a list of the objects contained in the shared environment.

Another problem stemming from shared components occurs when generating code for a programme. Unlike actions and conditions, which only appear in a process if explicitly referenced in an action flow or guard, every data object that is declared in an environment is included in a programme. As a consequence, if two processes are sharing an environment, and one process uses variable temp, then the code generated for both programmes will contain the declaration for variable temp. It is for these reasons that shared components are not allowed in this implementation.

6.2. THE STRUCTURE OF THE DATABASE TABLES

Though the databases that are used for each implementation are not identical, the differences between versions are minor. Perhaps one version has a table with just one index, while another version has the same table with more than one index. The size of an ID field might differ, or there might be a field, such as a time stamp, that appears in one version's action table, but not in another.

Another difference that exists from version to version is the use of tables or fields (or even variables which are kept on the disk), which help to facilitate "housekeeping" but do not have anything to do with the methodology. There might exist a system table that has such information as the last process that was in the system when the user exited, in order to allow the user to automatically continue with the same process on re-entering the editor.

The only change to the database that affects the ABL methodology has to do with the use of goals. In the new implementation, goals are not supported. The main reason that goals have not been supported is that they are just not needed. If the aim in using clusters as the only acceptable construct in programme design is to simplify programming, then the use of goals complicates things unnecessarily. One set of decision points suddenly becomes two, and the conditions that are tested at the level of the goal can easily be included in the guards for the alternatives of the next cluster.

Another reason for not including goals in the new ABL tool, is that they are rarely used. The ABL methodology has been used in software projects for over three years now, and in that time, almost all the projects were implemented without the use of goals. Furthermore, when new users first start using the ABL methodology, if given the choice, they would rather work without goals, because they find them confusing.

By examining the tables, and considering the relationships that exist between the tables in the database, one can see the underlying data model for the ABL methodology. For example, the action flow table relates the action table with the alternative table. The tables also show what objects belong to a process, and what objects belong to the strategy and the environment.

Without considering the goal table, because it is no longer being used, (and because it is identical in structure to the guard table), we see that the ABL tools, regardless of version, principally use the following tables :

Process table

Environment table

Strategy table

Cluster table

Alternative table

Action table

Condition table

Object table

Guard table

Action Flow table

We know that a process is composed of a strategy and an environment. The process table defines a process using an environment ID number and a strategy ID number. The environment and strategy tables describe the environment and strategy respectively, and the occurrence of their ID numbers in other tables defines their domain. The cluster table and the alternative table both have a strategy ID number. The action, condition, and object tables each have an environment ID number, and the guard and action flow tables have a process ID number.

Taken alone, the tables with a strategy ID number describe the top level view of a process. The strategy report, in fact, is composed of clusters and alternative descriptions. The alternative table completes this view by containing the source cluster ID number and the destination (next) cluster ID number associated with an alternative.

The tables with an environment ID number define the objects found in the environment, and the operations that can be performed on them. The object table contains the data object definitions, and the action table and condition table are composed of elements from the object table combined with their operators to form expressions.

It is the guard table and the action flow table which actually define the relationship between the strategy table and the environment table. The action flow table has an ID number for an alternative, an ID number for an action, and a position number. For example, an entry in the action flow table might have an alternative ID number of 5, an action ID number of 200, and a position number 3. This would mean that in this particular process, there existed an alternative 5 which had an action flow, such that, the third element of the action flow could be found by examining the action in the action table whose ID number was 200 (given the correct environment ID). The guard table works in a similar fashion, using conditions instead of actions, with the following changes. First of all, the conditions are not an ordered list and so the guard table does not have a field for the position number. However, it is necessary to keep track of the condition's value for the particular guard. The guard table has a value field to denote whether, in this instance, the condition referenced by the condition ID number, should be regarded as true or false (or yes or no).

One place where the new ABL tool differs from the old ABL tools is with regard to the use of ID numbers. In particular, in the older versions, the ID numbers for actions and conditions are used by the user. In fact, in the Oracle version, the only way to map the actions and the conditions to the alternatives is by filling in their ID numbers in the action flow and guard flow forms. In the dBASE III PLUS version, it is possible to avoid the use of ID numbers, but to do so entails such a convoluted process, that the ID number method of entry is really the only option available.

In the new version of the ABL tools, there still exist ID numbers for actions and conditions, but they are transparent to the user. The ID numbers are stored in the tables to maintain the relationships, to serve as a guide when loading a process into memory. However, they can be likened to addresses in machine code and are not seen, modified or used by the user. Getting rid of ID numbers is one step in creating an environment that allows development of software systems without the use of paper and pencil.

CHAPTER 7

CONCLUSION AND FUTURE CONSIDERATIONS

7.1. TOWARDS AN IMPLEMENTATION OF A COMPLETE SYSTEM

A prototype of new ABL tools with a syntax-driven full screen editor has been implemented. This prototype serves to demonstrate that it is possible to have full screen editing facilities for designing and implementing software systems using the ABL methodology. The editor makes it easier for the user to create ABL processes that adhere to the methodology, in a way that is unobtrusive to the user.

Testing of the prototype was conducted in an informal manner. People were asked to enter a small ABL process using the prototype and using SAM II. This simple test was sufficient in illustrating the value of a syntax-driven editor over a form oriented data entry system. Not only was it immediately apparent that the prototype was extremely fast and easy to use in comparison with SAM II, but in some cases, people were unable to enter and edit a simple ABL process using the SAM II system without extensive coaching.

The prototype currently has complete line editing and line navigating facilities, including the option of entering text in insert or replace mode. It supports full screen navigation, such as page up, page down, arrow up, and arrow down, as well as navigating to the top or bottom of an ABL process. It is possible to add or delete an ABL object, and all 'housekeeping', such as renumbering of ID numbers when an ABL object is added or deleted, is performed automatically.

The following functions are either incomplete or are completely missing from the prototype, and are necessary in tools that are used for building software systems.

7.1.1. ABL Reports

The prototype is capable of generating two reports, the strategy report and the process report. There are several reports that should be added here, the most important being the code report.

7.1.2. Block Commands

In order to share actions and conditions, block commands are needed. The block commands have been designed and coded, but have not been tested.

7.1.3. Code Editing

Though there is no working model of an editor in code mode, the structures in memory, in addition to supporting the narration mode, has support for the code mode. The prototype is capable of representing code using multiple lines which are variable length. The database has also been extended to allow for representing the code.

7.1.4 System Level Facilities

The editing facilities in the prototype operate only on the level of an ABL process. It is necessary to have access to details at the system level, such as the ABL processes and the design level specifications that comprise a system.

7.1.5. Analysis and Verification

The prototype already performs some analysis and verification on ABL processes, (for example, ensuring that a cluster exists if it is referenced in an alternative's next cluster field, and also, notifying the user when a cluster becomes orphaned), but these features only cover some of the possibilities. There exist other analysis and verification techniques that the ABL methodology can support, but which were not implemented because they do not fall under the realm of full screen editing.

7.2. FUTURE ENHANCEMENTS

So far the additions that have been proposed can be seen as features that are necessary in order to elevate the existing prototype to the status of a full fledged designing tool for large software systems. Assuming that these additions are implemented, there is still room for making improvements to the ABL tools.

7.2.1. Multiple Prccesses

It would be very convenient if the user could have access to more than one ABL process at a time. It would then be possible, for instance, to block off a cluster in one ABL process, and then toggle to another ABL process in order to copy the block.

7.2.2. Enhancements for Editing

Though the prototype for the ABL tools has full screen editing facilities, there are still some editing functions that are missing. For example, there is at present no way to make a global change. Most word processors allow the user to replace all (or some) occurrences of one string by another string.

There should also be a way for directly accessing ABL objects. The user should be able to specify a cluster or an alternative, at least by ID number if not by description, and have it become the current line.

7.2.3. Tracing the Path of an ABL Process

One enhancement which would be useful to include, would be a function that would allow the user to step through an ABL process. The user would choose an alternative from the current cluster, and the editor would automatically navigate to the cluster which is referenced by the alternative's next cluster field. In this way the user could perform a manual trace through the various paths that the ABL process would allow without actually executing the process.

7.2.4. Different Views of an ABL Process

Another feature that could be included in the ABL editor that would be helpful in designing an ABL process, would be to allow the user to choose various views of a process without having to generate a report. Currently, the present ABL editor only supports the standard strategy and process views. While this is sufficient for designing an ABL process, some people prefer to work with the tabular view. All the information that is needed for the tabular view is already present in the standard process view. It would only be necessary to work out the details of how to represent the information in a tabular format on the screen, as well as keeping track of which view is currently being displayed.

7.2.5. Multiple Users

The current ABL tools can only be used by one person at a time. Either people in a project take turns using the ABL tools or else each person in the software project has access to a computer running a separate copy of the ABL tools and its underlying database. An important feature that could be added in some future implementation of the ABL tools would be to allow several users to work concurrently. In this way it would also be easier for a project supervisor to monitor and control the progress of a software project.

7.2.6. Maintaining Existing Software Systems

The ABL tools help software experts design and implement new software systems. While this certainly fulfills a need in the software community, the ABL tools ignore a large segment of software engineering which is concerned with the problem of maintaining existing software systems. What would be instrumental in allowing the ABL tools to be able to work with these existing software systems, would be some utility that enables users to import non-ABL programmes into the ABL tools.

Some work has already been done in this direction. A small system was designed [7] to perform a static analysis of non-ABL programmes (written in Fortran), in order to extract control flow information. Using this control flow information, rudimentary clusters could be defined, and lines of code which were not able to be referenced could be identified. Another small system [8][10] was designed in order to partially automate the process of converting a large software system into the ABL format.

7.2.7 Data Considerations

It should come as no surprise that the ABL methodology excels in its ability to describe the control flow of a programme, because its strength lies in being able to specify algorithms. This does not mean that the ABL methodology has no means of dealing with the data flow and the data objects of an ABL process, but that the main emphasis of the ABL methodology is placed on the control flow of an ABL process. One need only consider that an ABL process is comprised of an environment as well as a strategy in order to attest to the fact that the data flow and data objects have been provided for in the ABL methodology.

There is, however, much work that can be done towards improving the ABL tools when it comes to operations that do not pertain to control flow. For instance, ensuring or verifying that every data object is defined, along with the drudgery of actually having to define each data object is something that the user must presently be responsible for. It is possible for future implementations of the ABL tools to be able to relieve the user of either part or all of this burden. There is also no facilities included in the ABL tools for designing a software system using data flow and data models.

A central data dictionary can be created by the ABL system. While the user is entering an ABL process using the ABL editor, it is possible to extract data objects using simple lexical analysis and parsing techniques. The user can then have access to the data dictionary in order to perform such functions as binding the data objects to a certain type. Though this strategy is very simple, and does not address all the needs of the user, it would still be a vast improvement over present facilities available to the user.

A more comprehensive approach to the problem of strengthening the ABL tools with respect to operations not involved with control flow, would have to include a facility that would allow the user to design a software system using data flow and data models. This could either be done by designing new data oriented facilities, or integrating existing data oriented tools into the ABL environment. The ABL design philosophy is flexible enough to be used in conjunction with other software development tools and methodologies. This approach, either by itself or combined with the central data dictionary option, will make the ABL tools much more comprehensive.

Ultimately more data facilities will be added to the ABL tools. The result will be that the user will have a more powerful and more flexible set of tools to work with. More importantly, the addition of data flow and data model facilities will enable the ABL tools to come closer to being a complete CASE environment for designing and implementing large software systems, whether these software systems fall into the domain of real-time and process control, or business and MIS applications.

References

1. Ashton-Tate. **Using dBASE III Plus reference manual.** Ashton-Tate.

2. Auto-Asyst software package by Atkinson-Trembly.

3. CASE 2000 software package by Nastec Inc.

4. Chvalovsky V. "Decision tables". **Software - practice and Experience** 13 (1983). pp.423 - 429.

5. Consultech Canada. "Proposal for SAM III". **Internal document of Syslog Inc.** Montreal.(1987).

6. ITC. "A guided tour of Excelerator". **Index Technology Corp.** Cambridge, MA. (1985).

7. Finkelstein K. "A design for a static analyzer". **A project for the graduate course "Computer-aided Design and Research"** at Concordia University. (1986).

8. Finkelstein K. "Programmer's Guide for SAM II Import Utility for Programming Languages". **Internal document of Syslog Inc.** Montreal. (Feb. 1987).

9. Finkelstein K. "The Telecommande en Ligne project". **Internal document of Syslog Inc.** Montreal. (1987).

10. Finkelstein K. "User Guide for SAM II Import Utility for Programming Languages". **Internal document of Syslog Inc.** (Feb. 1987).

11. Hamilton M. and Zeldin S. "Higher Order Software: A methodology for defining software". **IEEE Trans. Soft. Eng.** SE-2: 1 (Mar. 1976) pp.71 - 79.

12. IDE software package by Interactive Development Environments.

13. IEW (Information Engineering Workbench) software package by KnowledgeWare Inc.

14. Jaworski W., Ficocelli L., and O'Mara K. "ABL/W4, A language-independent environment for software science". **Concordia University.** Montreal. (1983).

15. Jaworski W. and Virard M. "Converting a software company to a new technology". **Proceedings of the 1986 Canadian Conference on Industrial Computer Systems.** Montreal. (1986). pp.12-1 - 12-7.

16. Jaworski W. "Computer-aided design and research course notes". **Concordia University.** Montreal. (1986).

17. Jaworski W., MacCuaig I., Marinelli T., and Nyisztor T. " 'Executable' specification for a large industrial process". **Proceedings of the 1986 Canadian Conference on Industrial Computer Systems.** Montreal. (1986). pp.60-1 - 60-5.

18. Jaworski W. M. "Introduction to ABL/W4: A software development system". Research Report **Concordia University.** Montreal. (1984).

19. Jaworski W., Ficocelli L., and O'Mara K. "The ABL/W4 approach: A view of representational distortion, documentation and software pragmatics". Research Report **Concordia University.** Montreal. (1984).

20. Lejderman J. "CASE: The technology transfer issue". **Computing Canada.** (July 1987). pp.18 - 19.

21. W. M. Jaworski and H. Hinterberger. "Controlled Program Design by use of the ABL Programming Concept". **Angewandte Informatik** (Applied Mathematics). Wiesbaden, Germany, 302-310. (1981).

22. McMullen W. structured decision tables". **SIGPLAN Notices** 19: 4. (Apr. 1984). pp.34 - 43.

23. Myers H. J. "Compiling optimized code from decision tables". **IBM Journal of Research and Development** 16: 5 (Sept. 1972). pp.489 - 503.

24. Oracle Corporation. "Database Administrator's Guide". **Oracle Corporation.** (1984).

25. ProMod software package by ProMod Inc.

26. Syslog Inc. "SAM II User's Manual vers. 1.22". **Syslog Inc.** Montreal (Dec. 1987).

27. Robillard P. "Schematic Pseudocode for program constructs and its computer automation by SCHEMACODE". **Communications of the ACM** 29: 11 (Nov. 1986). pp.1072 - 1089.

28. Small C. "Automated decision logic verification in a CASE system". **Internal document of Syslog Inc.** Montreal (1987).

29. Small C. "Circuit board testing system converted using SAM tools". **Internal document of Syslog Inc.** Montreal (1986).

30. Small C. "Software conversion using an automated development methodology". Technical Proceedings HP1000/ 9000. **Proceedings of the 1986 INTEREX Conference** (Detroit, Sept. 28 - Oct. 3, 1986). International Association of Hewlett-Packard Computer Users. pp.1003-1 - 1003-29.

31. Sommerville I. **Software Engineering.** Addison-Wesley, London. (1982).

32. SOS software package by Syslog Inc. Montreal

33. Suydam W. "CASE makes strides toward automated software development". **Computer Design.** (January 1, 1987). pp.49-70.

34. Syslog Inc. "An SOS Glossary". **in seminar on Syslog Automation Methodology.** Syslog Inc. Montreal. (1986).

35. Teamwork software package by CADRE Technologies.

36. Vick C. R. "A software engineering environment". **Handbook of Software Engineering.** C.R. Vick and C.V. Ramamoorthy, Eds. Van Nostrand Reinhold Company. New York. (1984).

37. Virard M. A. "Transfer of Engineering knowledge". **Engineering Digest.** (Sept. 1986). pp.32 - 33.

38. Wirth N. **Systematic Programming: An Introduction.** Prentice-Hall, N.J. (1973).

39. Small Charles. "Position Paper". **First International Workshop on Computer-Aided Software Engineering.** Cambridge, Massachusetts. (May 27 - May 29) 1987.

APPENDIX 1

ABL GLOSSARY

ACTION

Actions are constituents of environments. They are of three types: compound, simple and complex.

A compound action may be decomposed into a non-empty ordered sequence of actions, where the order defines the sequence of execution of the actions. All actions referenced in a compound action must themselves be constituents of the same environment as the compound action. The constituent actions may themselves be of any type: compound, simple, or complex. Compound actions are not primitive, in the sense that they are not associated with source code in an implementation language.

Simple actions are not decomposable into other actions. They correspond with and are associated with source code in an
implementation language -- typically, executable actions or sequences of them (In practice, simple actions may be translated into implementation language code which contains loops or condition tests, as well). A simple action can be seen as composed of primitive objects in the environment of the action together with functional operators which access or manipulate the values of those objects.

Complex actions, like simple actions, are not decomposable into other actions. They are also associated with source language code -- in this case, typically, procedure calls. Execution of a complex action corresponds to the invocation of another process, with its own strategy and environment, at the point at which the complex action is executed. In the context of the environment in which it occurs, the complex action which corresponds to a process appears as a "black box". This facility allows systems to be structured as hierarchies of processes.

ACTION FLOW

Action flows are defined with respect to processes, since they relate alternatives (of a strategy) with actions (of an environment). The action flow for an alternative is an ordered sequence (possibly empty) of actions, each of which must be defined in the environment of the process with which the alternative is associated.

Action flows define, for an alternative, the sequence in which the actions associated with it are to be carried out, when the alternative is chosen for execution.

An action may occur any number of times within a given action flow -- that is, the same action may be carried out more than once for a given alternative, at different points in the sequence of execution of the alternative's actions.

ALTERNATIVE

Alternatives are (along with clusters) the constituents of strategies. Each strategy may have any number of alternatives (although in practice strategies with more than 25 or so
alternatives tend to become unwieldy). Each alternative is associated with:

a) an alternative number, unique within its strategy;
b) the number of the cluster within its strategy which is the STARTING cluster for the alternative;
c) the number of the cluster within its strategy which is the NEXT cluster for t . alternative;
d) (optionally) the number of a cluster within its strategy which is the EXCEPTION cluster for the alternative.

An alternative defines a path from one decision point to another. The path ordinarily goes from STARTING cluster to NEXT cluster, but may also go from STARTING cluster to EXCEPTION cluster.

During execution, the alternatives starting at a cluster define the options, or possible paths, to be taken from that point. Where (as is normally the case) there is only a single non-concurrent processor, only one path from a cluster must be chosen.
Implicitly, some set of conditions must be evaluated to determine which path to take. Implicitly, also, taking the path means executing some sequence of actions. However, these conditions and actions are not defined in the strategy, but in the environment of a process with which it is associated.

Where the execution of the actions of an alternative is successful, execution will proceed to the NEXT cluster. Where it is not, execution will go to the EXCEPTION cluster for error handling. Success of an alternative is determined by evaluation of another set of conditions known as GOALS, which are also defined in the associated environment.

Within a strategy, alternatives are ordinarily numbered sequentially from 1. However, the assignment of numbers is arbitrary. That is, the numbers need not be consecutive, and no particular scheme of sequential assignment is required.

CLUSTER

Clusters are one of the components of strategies. Each strategv may have any number of clusters (although in practice strategies with more than eight or so clusters tend to become unwieldy). Each cluster is a decision point which may be designated as the STARTING cluster, NEXT cluster or EXCEPTION cluster for any of the alternatives of the strategy.

During execution, when each cluster is reached, the values of the conditions associated with the alternatives which start at it are tested, and a single alternative is chosen for execution from among those starting at the cluster. By convention, clusters are numbered sequentially within each strategy, starting from 1. Cluster 1 is an unconditional start point (process entry) and cluster 0 an unconditional stop point (process exit or termination). Cluster 0 can never appear as a starting cluster.

CONDITION

Conditions are components of environments. They are of two types: compound and simple.

A compound condition may be decomposed into a (possibly empty) ordered sequence of actions (where the order defines the sequence of execution of the actions), followed by a single simple condition. All actions and conditions referenced in a compound condition must themselves be constituents of the same environment as the compound condition. The constituent actions may themselves be of any type: compound, simple, or complex. Compound conditions are not primitive, in the sense that they are not associated with source code in an implementation language.

Simple conditions are not decomposable into conditions and actions. They correspond with and are associated with source code in an implementation language -- typically, conditions to be tested. A simple condition may be seen as composed of primitive objects in the environment of an action together with functional operators which access the values of those objects and relational operators which test them. Ultimately, a simple condition returns a single Boolean value (T or F). It may contain embedded functional operators used in evaluation (+, -, *, etc.) provided no values of objects are altered by these operators. Simple conditions may in practice contain arbitrary parenthesized combinations of ANDs and ORs. Conventional rules govern the priority of the evaluation of these.

ENVIRONMENT

An environment is the data flow component of a Process. It is represented as a set of Objects, a set of Actions, and a set of Conditions. Simple Actions and Conditions can themselves be decomposed into Objects and Operators which act on them (in the case of Actions) or test their values (in the case of Conditions). Actions may also be complex, in which case they correspond to the invocation of another Process with its own Strategy and Environment at the point at which the Action is executed.

An environment may be created and edited independently of any process or strategy, but its components can only be accessed and activated during execution by a process with which it is associated.

GUARD

A guard is a function of the interaction of a strategy with an environment, and is therefore defined with respect to a process. Guards define, for a process, the circumstances under which a particular path will be chosen at a decision point (or, in other words, the circumstances under which a particular alternative will be selected for execution at a cluster). Each guard is associated with:

a) an alternative in the strategy of its process;

b) a condition in the environment of its process;

c) a required value (T or F) for that condition.

Each alternative of a strategy may be associated with 0 or more guards. During execution, all the guards for all alternatives starting at the current cluster are selected, and, for each guard, the current value of its condition is compared with the required value (T or F) in the guard. If the current and required values of all guards for an alternative match, that alternative is selected for execution.

Guard values must be chosen in such a way as to ensure that in all cases one and only one alternative starting at a cluster will be chosen for execution. This implies that the guards for a cluster define a complete, consistent and mutually exclusive set of alternatives.

GOAL

A goal, like a guard, is a function of the interaction of a strategy with an environment, and is defined with respect to a process. Goals define, for an alternative, the circumstances under which execution of the actions of that alternative will be considered successful and when it will be considered unsuccessful. This evaluation determines, in turn, whether execution will proceed to the NEXT cluster for the alternative or the EXCEPTION cluster.

Each goal is associated with:

a) an alternative in the strategy of its process;
b) a condition in the environment of its process;
c) a required value (T or F) for that condition.

Each alternative of a strategy may be associated with 0 or more goals. If no goals are present, execution always proceeds to the NEXT cluster. Otherwise, for each goal, the current value of its condition is compared with the required value (T or F) in the goal. If the current and required values of all goals for an alternative match, execution proceeds to the NEXT cluster. If any condition does not have the value required for it in the goal, execution proceeds to the EXCEPTION cluster.

OBJECT

Objects are of three types: complex, simple and primitive.

Complex objects are components of systems and may be decomposed into simple objects and other complex objects. They are not associated with values and are not represented by source language code in any specific environment. They do appear, however, as entries in the global data dictionary for the system.

Simple objects, like complex objects, appear as entries in the global data dictionary and are not associated with values or represented by source language code in any environment. However, unlike complex objects, they are not decomposable. Each simple object in the data dictionary corresponds to at least one primitive object in some environment of the system. It may in fact correspond to objects in many different system environments. In each of these it will be classified as either of type INPUT or type OUTPUT. A simple object may not correspond to more than one object in any given environment.

A simple object which corresponds to a primitive object in only one environment is said to be LOCAL to that environment. A simple object which corresponds to primitive objects in more than one environment is said to be GLOBAL to all those environments.

Primitive objects are components of specific environments. They do not appear in the global data dictionary, and are not decomposable. Each primitive object, however, corresponds to some particular simple object in the global data dictionary. Primitive objects are associated with values or sets of values of defined types. Like actions and conditions, they are represented by source language code (object declarations, typically) in some implementation language. Primitive objects may be combined using functional operators (+, -, *, /, assignment, MOD, max, exponential, square root, etc.) to produce actions or relational operators (>, <, >=, <=, =, <>, AND, OR, NOT) to produce conditions.

PROCESS

A process is a unit of execution within a system. It is the execution of a strategy within an environment. A process begins executing when it is invoked, either by a user or by another process. It begins at its entry point (by convention, cluster 1) and continues until cluster 0 (the exit point) is reached.

STRATEGY

A strategy is the control flow component of a process. It is represented as a set of clusters and a set of alternatives, where the clusters represent decision points and the alternatives represent paths connecting the decision points. A strategy may be created and edited independently of any process or environment, but can only be executed by a process with which it is associated.

SYSTEM

A system is a network of related processes. The processes in a system may be:

a) Hierarchically related, with a single thread of execution. b) Communicating processes with multiple threads of execution. c) Processes related in neither of these ways but only
conceptually.
d) Combinations of a), b), and c).

APPENDIX 2

ABL REPORTS

The following pages contain samples of ABL reports. The first report is the code report, which has the detailed design level embedded in it.

The second report is a strategy report. It is followed by a segment of a process report and a tabular report.

```
#include <stdio.h>
#include <stdlib.h>                          /* malloc
#include <string.h>                          /* strcmp,strncpy,strcat

#include "global.h"
#include "scrnglbl.h"
#include "extern.h"

int bldactbl(penv,actb,acte)
char penv[IDSIZE + 1];                       /* environment id for actions
struct actntbl **actb;                       /* beginning of action table
struct actntbl **acte;                       /* end of action table
{
/*

        file bldactbl.c         Written by K. Finkelstein

        LAST MODIFIED : Sept 14 1988.

        - Dynamic code field added

        This function builds a table of actions for the requested
        environment - the table has a pointer at its beginning and
        end (actb and acte).

        ASSUMPTIONS    :  Action 1 exists (if any actions for this env.).
                          Actions are ordered 1 .. n.
                          Actions are reused when saved; i.e., if action x
                          is INACTIVE then there is no action x.
                          The above assumption translates to mean that if
                          action 1 is INACTIVE then there are no actions.

        RETURN STATUS :   0  SUCCESS
                          1  Out of Memory error (Warning)
                         -1  Database Error      (Fatal)
                         -2  No actions exist    (Warning)

*/
#include <dbc.h>                             /* dbc routines, variables
#include "action.h"                          /* action table

extern dBopen(),dBiopen(),dBgetnr(),dBgetrk(),dBclose(),dBiclose();

#define ACT1 "    0001"                      /* action 1
#define BGNENV  0                            /* start of environment (strinso)
#define WHOLE 0                              /* type whole for asci_to_number

/****************************************************************************
 1 Initialize

   A1 Initialize                                                      ->  2

 2 Open Index file

   A2 Open Index file                                                 ->  3
   A3 Error opening database file                                     ->  0

 3 Get First Action
```

```
   A4 Get first action                                              ->  4
   A5 Error opening index file                                      ->  0

 4 Get Memory for First Entry

   A6 Get memory for Top                                            ->  5
   A7 Deleted Action                                                ->  0
   A8 No action exists                                              ->  0
   A9 Error reading database                                        ->  0

 5 Initialize First Action

   A10 Initialize first action                                      ->  6
   A11 No memory available                                          ->  0

 6 Get Environment from record

   A12 Get environment from record                                  ->  7
   A13 Deleted Action                                               ->  6
   A14 End of actions                                               ->  0
   A15 Error reading database                                       ->  0

 7 Get memory for entry

   A16 Get memory for entry                                         ->  8
   A17 End of actions                                               ->  0

 8 Initialize Current Action

   A18 Initialize current action                                    ->  6
   A19 No memory available                                          ->  0
******************************************************************************

struct actntbl *tmptr;                   /* temporary pointer for new nodes
struct actntbl *curptr;                  /* current pointer in action table
struct xactlst *item;                    /* current pointer for action's code
char key[ACTENVLT + ACTACTLT + 1];       /*  key env + act
char status;                             /*  record status: ACTIVE or INACTIVE
char *tmpcode;                           /*  temp. ptr - holds code for action
int error;                               /*  dbc routines status variable
int i_num;                               /* temp. integer for asci_to_number
int match;                               /*  status of string comparison
int rcode;                               /* return status for function
double f_dummy;                          /*  dummy double for asci_to_number

Clus01:
/*****************************************************************************
C1 Initialize

   A1 Initialize
        Clear Pointers
        Open database file
                        next 2

******************************************************************************
```

```
/* A1 Initialize
   *actb = NULL;
   *acte = NULL;
   error = dBopen(DBACTION,&Daction);


Clus02:
/*****************************************************************************
C2 Open Index file

   A2 Open Index file
      IF successful database access
       Open index file
                        next 3

   A3 Error opening database file
      IF NOT successful database access
        Handle error
        Set return status
                        next 0

*****************************************************************************

if (error == SUCCESS)
  {
/* A2 Open Index file
   error = dBiopen(IXACTION,&Iaction);
   goto Clus03;
  }
else
  {
/* A3 Error opening database file
   errhndlr(14);
   rcode = -1;
   goto Clus00;
  }


Clus03:
/*****************************************************************************
C3 Get First Action

   A4 Get first action
      IF successful database access
        Construct key for indexed read
        Read action from database (indexed)
                        next 4

   A5 Error opening index file
      IF NOT successful database error
        Handle error
        Close database file
        Set return status
                        next 0

*****************************************************************************

if (error == SUCCESS)
```

```
  {
/* A4 Get first action
   strasgn(ACT1,key);
   strinso(key,BGNENV,IDSIZE,penv);
   error = dBgetrk(Daction,Iaction,key,Actrec,&status);
   goto Clus04;
  }
else
  {
/* A5 Error opening index file
   errhndlr(15);
   error = dBclose(Daction);
   if (error != SUCCESS) errhndlr(18);
   rcode = -1;
   goto Clus00;
  }


Clus04:
/*****************************************************************************
C4 Get Memory for First Entry

   A6 Get memory for Top
      IF successful database access
     AND ACTIVE record
        Get memory (First)
        Get memory for action code
                         next 5

   A7 Deleted Action
      IF NOT successful database access
     AND NOT ACTIVE record
        Close  database file
        Close index file
        Set return status
                         next 0

   A8 No action exists
      IF no record
        Close  database file
        Close index file
        Set return status
                         next 0

   A9 Error reading database
      IF NOT successful database access
      AND NOT no record
        Handle error
        Close  database file
        Close index file
        Set return status
                         next 0

*****************************************************************************

 if (error == SUCCESS)
   {
    if (status == ACTIVE)
```

```
        {
/*       A6 Get memory for Top
         *actb = (struct actntbl *) malloc(sizeof(struct actntbl));
         item = (struct xactlst *) malloc(sizeof(struct xactlst));
         goto Clus05;
        }
      else
        {
/*       A7 Deleted Action
         error = dBclose(Daction);
         if (error != SUCCESS) errhndlr(18);
         error = dBiclose(Iaction);
         if (error != SUCCESS) errhndlr(19);
         rcode = -2;
         goto Clus00;
        }
    }
else
    {
      if ((error == d_NOKEY) || (error == d_ENDKEY))
        {
/*       A8 No action exists
         error = dBclose(Daction);
         if (error != SUCCESS) errhndlr(18);
         error = dBiclose(Iaction);
         if (error != SUCCESS) errhndlr(19);
         rcode = -2;
         goto Clus00;
        }
      else
        {
/*       A9 Error reading database
         errhndlr(16);
         error = dBclose(Daction);
         if (error != SUCCESS) errhndlr(18);
         error = dBiclose(Iaction);
         if (error != SUCCESS) errhndlr(19);
         rcode = -1;
         goto Clus00;
        }
    }


Clus05:
/*************************************************(.***************************
C5 Initialize First Action

   A10 Initialize first action
       IF memory available
        Set extended action to NIL
        Set code to NIL
        Attach code pointer to action table
        Get code
        Get blanks
        Copy code without blanks
        If successful attach code to action table
        Get narration
        Get id number (Ascii to Integer)
        Initialize current pointer to action
        Initialize bottom of table to action
        Set next to NIL
        Set previous to NIL
        Read next action from database
                        next 6

   A11 No memory available
        IF NO memory available
         Handle error
         Close  database file
         Close index file
         Set return status
                        next 0

****************************************************************************

if ((*actb != NULL) && (item != NULL) )
  {
/* A10 Initialize first action
   item->next = NULL;
   item->cod = NULL;
   (*actb)->xact = item;
   error = gtsbstr(Actrec,ACTCODST,ACTCODLT,Act_cod);
   if (error != SUCCESS) errhndlr(17);
   asci_to_number(Actrec,ACTBLNST,ACTBLNLT,WHOLE,&Act_bln,&f_dummy);
   error = gtdynstr(Act_cod,&tmpcode,ACTCODLT,Act_bln);
   if (error == SUCCESS) ((*actb)->xact)->cod = tmpcode;
   error = gtsbstr(Actrec,ACTNARST,ACTNARLT,(*actb)->nar);
   if (error != SUCCESS) errhndlr(17);
   asci_to_number(Actrec,ACTACTST,ACTACTLT,WHOLE,&i_num,&f_dummy);
   (*actb)->id = i_num;
   curptr = *actb;
   *acte = *actb;
   (*acte)->dwn = NULL;
   (*acte)->up = NULL;
   error = dBgetnr(Daction,Iaction,Actrec,&status);
   goto Clus06;
  }
else
  {
/* A11 No memory available
   errhndlr(4);
   error = dBclose(Daction);
   if (error != SUCCESS) errhndlr(18);
   error = dBiclose(Iaction);
   if (error != SUCCESS) errhndlr(19);
   rcode = 1;
   goto Clus00;
  }


Clus06:
/****************************************************************************
C6 Get Environment from record

   A12 Get environment from record
       IF successful database access
```

```
      AND ACTIVE record
        Get environment from record
                        next 7

   A13 Deleted Action
      IF successful database access
      AND NOT ACTIVE record
        Read next action from database
                        next 6

   A14 End of actions
      IF end of database
        Close  database file
        Close index file
        Set return status
                        next 0

   A15 Error reading database
      IF NOT successful database access
      AND NOT end of database
        Handle error
        Close  database file
        Close index file
        Set return status
                        next 0

****************************************************************************

if (error == SUCCESS)
  {
   if (status == ACTIVE)
     {
/*    A12 Get environment from record
      error = gtsbstr(Actrec,ACTENVST,ACTENVLT,Act_env);
      if (error != SUCCESS) errhndlr(17);
      match = strcmp(penv,Act_env);
      goto Clus07;
     }
   else
     {
/*    A13 Deleted Action
      error = dBgetnr(Daction,Iaction,Actrec,&status);
      goto Clus06;
     }
  }
else
  {
   if ((error == d_NOKEY) || (error == d_ENDKEY))
     {
/*    A14 End of actions
      error = dBclose(Daction);
      if (error != SUCCESS) errhndlr(18);
      error = dBiclose(Iaction);
      if (error != SUCCESS) errhndlr(19);
      rcode = 0;
      goto Clus00;
     }
   else
```

```
     {
/*    A15 Error reading database
      errhndlr(16);
      error = dBclose(Daction);
      if (error != SUCCESS) errhndlr(18);
      error = dBiclose(Iaction);
      if (error != SUCCESS) errhndlr(19);
      rcode = -1;
      goto Clus00;
     }
  }

Clus07:
/*****************************************************************************
C7 Get memory for entry

   A16 Get memory for entry
      IF action in program environment
       Get memory
       Get memory for action code
                       next 8

   A17 End of actions
      IF NOT action in program environment
        Close  database file
        Close index file
        Set return status
                       next 0

*****************************************************************************

if (match == SUCCESS)
  {
/* A16 Get memory for entry
   tmptr = (struct actntbl *) malloc(sizeof(struct actntbl));
   item = (struct xactlst *) malloc(sizeof(struct xactlst));
   goto Clus08;
  }
else
  {
/* A17 End of actions
   error = dBclose(Daction);
   if (error != SUCCESS) errhndlr(18);
   error = dBiclose(Iaction);
   if (error != SUCCESS) errhndlr(19);
   rcode = 0;
   goto Clus00;
  }


Clus08:
/*****************************************************************************
C8 Initialize Current Action

   A18 Initialize current action
       IF memory available
        Set extended action to NIL
        Set code to NIL
        Attach code pointer to action table
        Get code
        Get blanks
        Copy code without blanks
        If successful attach code to action table
        Get narration
        Get id number (Ascii to Integer)
        Set current pointer -> dwn to temp. pointer
        Set temp. pointer -> up to current pointer
        Initialize current pointer to action
        Initialize bottom of table to action
        Set next to NIL
        Read next action from database
                        next 6

   A19 No memory available
       IF NO memory available
        Handle error
        Close  database file
        Close index file
        Set return status
                        next 0

***************************************************************************

if ( (tmptr != NULL) && (item != NULL) )
  {
/* A18 Initialize current action
   item->next = NULL;
   item->cod = NULL;
   tmptr->xact = item;
   error = gtsbstr(Actrec,ACTCODST,ACTCODLT,Act_cod);
   if (error != SUCCESS) errhndlr(17);
   asci_to_number(Actrec,ACTBLNST,ACTBLNLT,WHOLE,&Act_bln,&f_dummy);
   error = gtdynstr(Act_cod,&tmpcode,ACTCODLT,Act_bln);
   if (error == SUCCESS) (tmptr->xact)->cod = tmpcode;
   error = gtsbstr(Actrec,ACTNARST,ACTNARLT,tmptr->nar);
   if (error != SUCCESS) errhndlr(17);
   asci_to_number(Actrec,ACTACTST,ACTACTLT,WHOLE,&i_num,&f_dummy);
   tmptr->id = i_num;
   curptr->dwn = tmptr;
   tmptr->up = curptr;
   curptr = tmptr;
   *acte = tmptr;
   tmptr->dwn = NULL;
   error = dBgetnr(Daction,Iaction,Actrec,&status);
   goto Clus06;
  }
else
  {
/* A19 No memory available
   errhndlr(4);
   error = dBclose(Daction);
   if (error != SUCCESS) errhndlr(18);
   error = dBiclose(Iaction);
   if (error != SUCCESS) errhndlr(19);
   rcode = 1;
   goto Clus00;
  }

Clus00:
/*****************************************************************************
Exit with Status

******************************************************************************
return(rcode);
}
```

```
                         Strategy Report

        This function builds a table of actions for the environment

          PRC : bldactbl        STR : bldactbl        ENV : bldactbl
-----------------------------------------------------------------------------
1    Initialize


     A1    Initialize                                          ->   2

2    Open Index file

     A2    Open Index file                                     ->   3
     A3    Error opening database file                         ->   0

3    Get First Action

     A4    Get first action                                    ->   4
     A5    Error opening index file                            ->   0

4    Get Memory for First Entry

     A6    Get memory for Top                                  ->   5
     A7    Deleted Action                                      ->   0
     A8    No action exists                                    ->   0
     A9    Error reading database                              ->   0

5    Initialize First Action

     A10   Initialize first action                             ->   6
     A11   No memory available                                 ->   0

6    Get Environment from record

     A12   Get environment from record                         ->   7
     A13   Deleted Action                                      ->   6
     A14   End of actions                                      ->   0
     A15   Error reading database                              ->   0

7    Get memory for entry

     A16   Get memory for entry                                ->   8
     A17   End of actions                                      ->   0

8    Initialize Current Action

     A18   Initialize current action                           ->   6
     A19   No memory available                                 ->   0
```

```
                         Process Report

         This function builds a table of actions for the environment

          PRC : bldactbl        STR : bldactbl        ENV : bldactbl
----------------------------------------------------------------------------
1    Initialize

     A1    Initialize
             Clear Pointers
             Open database file
                      NEXT 2


2    Open Index file

     A2    Open Index file
           IF     successful database access
             Open index file
                      NEXT 3

     A3    Error opening database file
           IF NOT successful database access
             Handle error
             Set return status
                      NEXT 0


3    Get First Action

     A4    Get first action
           IF     successful database access
             Construct key for indexed read
             Read action from database (indexed)
                      NEXT 4

     A5    Error opening index file
           IF NOT successful database error
             Handle error
             Close database file
             Set return status
                      NEXT 0


4    Get Memory for First Entry

     A6    Get memory for Top
           IF successful database access
          AND ACTIVE record
             Get memory (First)
             Get memory for action code
                      NEXT 5

     A7    Deleted Action
           IF NOT successful database access
          AND NOT ACTIVE record
```

```
                                  Tabular Report

                 This function builds a table of actions for the environment

                  PRC : bldactbl        STR : bldactbl        ENV : bldactbl
--------------------------------------------------------------------------------

CLUSTERS
       A1  A2  A3  A4  A5  A6
C 1.    X   .   .   .   .   .   .   .   C 1  Initialize
C 2.    .   X   X   .   .   .   .   .   C 2  Open Index File
C 3.    .   .   .   X   X   .   .   .   C 3  Get First Action
C 4.    .   .   .   .   .   X   .   .   C 4  Get Memory For First Entry
CONDITIONS
B 1.    .   T   F   T   F   T   .   .   B 1  successful database access
B 2.    .   .   .   .   .   T   .   .   B 2  ACTIVE record
B 3.    .   .   .   .   .   .   .   .   B 3
B 4.    .   .   .   .   .   .   .   .   B 4
B 5.    .   .   .   .   .   .   .   .   B 5
B 6.    .   .   .   .   .   .   .   .   B 6
B 7.    .   .   .   .   .   .   .   .   B 7
B 8.    .   .   .   .   .   .   .   .   B 8
B 9.    .   .   .   .   .   .   .   .   B 9
B10.    .   .   .   .   .   .   .   .   B10
B11.    .   .   .   .   .   .   .   .   B11
B12.    .   .   .   .   .   .   .   .   B12
ACTIONS
A 1.    1   .   .   .   .   .   .   .   A 1  Clear Pointers
A 2.    2   .   .   .   .   .   .   .   A 2  Open database file
A 3.    .   1   .   .   .   .   .   .   A 3  Open index file
A 4.    .   .   1   .   1   .   .   .   A 4  Handle error
A 5.    .   .   2   .   3   .   .   .   A 5  Set return status
A 6.    .   .   .   1   .   .   .   .   A 6  Construct key for indexed read
A 7.    .   .   .   2   .   .   .   .   A 7  Read action from database (indexe
A 8.    .   .   .   .   2   .   .   .   A 8  Close database file
A 9.    .   .   .   .   .   1   .   .   A 9  Get memory (First)
A10.    .   .   .   .   .   2   .   .   A10  Get memory for action code
A11.    .   .   .   .   .   .   .   .   A11
A12.    .   .   .   .   .   .   .   .   A12
A13.    .   .   .   .   .   .   .   .   A13
A14.    .   .   .   .   .   .   .   .   A14
A15.    .   .   .   .   .   .   .   .   A15
A16.    .   .   .   .   .   .   .   .   A16
A17.    .   .   .   .   .   .   .   .   A17
A18.    .   .   .   .   .   .   .   .   A18
A19.    .   .   .   .   .   .   .   .   A19
A20.    .   .   .   .   .   .   .   .   A20
A21.    .   .   .   .   .   .   .   .   A21
A22.    .   .   .   .   .   .   .   .   A22
A23.    .   .   .   .   .   .   .   .   A23
A24.    .   .   .   .   .   .   .   .   A24
A25.    .   .   .   .   .   .   .   .   A25


N   .   2   3   0   4   0   5   .   .   N
```