INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Kiper, James Dennis

THE ERGONOMIC, EFFICIENT, AND ECONOMIC INTEGRATION OF EXISTING TOOLS INTO A SOFTWARE ENVIRONMENT

The Ohio State University

Ph.D. 1985

University Microfilms International

Copyright 1985
by
Kiper, James Dennis
All Rights Reserved
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages 
2. Colored illustrations, paper or print 
3. Photographs with dark background 
4. Illustrations are poor copy 
5. Pages with black marks, not original copy 
6. Print shows through as there is text on both sides of page 
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements 
9. Tightly bound copy with print lost in spine 
10. Computer printout pages with indistinct print 
11. Page(s) lacking when material received, and not available from school or author.
12. Page(s) 102 seem to be missing in numbering only as text follows.
13. Two pages numbered . Text follows.
14. Curling and wrinkled pages 
15. Dissertation contains pages with print at a slant, filmed as received 
16. Other 

__________________________________________
__________________________________________

University Microfilms International
THE ERGONOMIC, EFFICIENT, AND ECONOMIC
INTEGRATION OF EXISTING TOOLS
INTO A SOFTWARE ENVIRONMENT

Dissertation

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

James Dennis Kiper, B.A., M.S., M.S.

The Ohio State University

1985

Reading Committee:
Dr. Jayashree Ramanathan
Dr. Sandra Mamrak
Dr. Venkataraman Ashok

Approved By

Dr. Jayashree Ramanathan
Adviser
Department of Computer
and Information Science
Dedicated to my wife, Beth
for her Eternal Patience and Love
Acknowledgement

My advisor, Professor Jay Ramanathan, receives my gratitude for her help and patience in guiding my work. Her words of encouragement have often come at important points when my endurance was failing. Overcoming a fifty mile distance obstacle has required her patience when my presence was needed but not readily available.

I would also like to thank the members of my reading committee, Professors Sandy Mamrak and Venkataraman Ashok, for their suggestions and corrections in the preparation of this dissertation.

I wish to thank members of the TRIAD project, particularly Walter McKnight. Colleagues are an important source of information and verification of ideas. Walter McKnight has contributed much of a technical nature to my implementation and to the preparation of the dissertation. His friendliness in giving this aid will always be remembered.

I wish also to thank Dr. Chin Li, Dr. Jeremy Kuo, and Dr. Dilip Soni for their contribution to my success by making me feel welcomed in the TRIAD group and in explanations of their ideas upon which I have built.

Most importantly, I thank God for an understanding of the purpose
and importance of my work, and my wife for the patience to endure the pressure brought by many tasks and for the understanding to encourage me during disappointments.
Vita

EDUCATION

June 13, 1953 Born, Burlington, Iowa

June 1975 B. S. Mathematics and Psychology, Olivet Nazarene College

June 1978 M.S. Mathematics, The Ohio State University, Columbus, Ohio

August 1978 M.S. Computer Science, The Ohio State University, Columbus, Ohio

WORK EXPERIENCE

Chairman, Department of Mathematics and Computer Science, Mount Vernon Nazarene College, Mount Vernon, Ohio, from August, 1982 to the present.

Assistant Professor of Mathematics and Computer Science Mount Vernon Nazarene College, Mount Vernon, Ohio, from August, 1982 to the present.


Lecturer in Computer Science, summer quarter 1982, in the Department of Computer and Information Science at the
Ohio State University, Columbus, Ohio.

Programmer at the Marathon Oil company executive offices in Findlay, Ohio, during the summer of 1981.

Teaching Assistant in the Department of Mathematics at the Ohio State University, Columbus, Ohio, from June, 1975, to August, 1978.
Research Interests

1. Software Environments: ergonomic interface design, human factors analysis of various environmental features.

2. Software Development Tools: the development new software tools for all phases of the software development cycle, the extension of the capabilities of existing tools, human factors analysis of ergonomic features of tools, the integration of tools, the development and analysis of incremental tools other than incremental compilers.

3. Artificial Intelligence: the application of Artificial Intelligence techniques to the development of software environments, the construction of an expert system for some limited programming domain.

4. Database Systems: the use of a relational database system to store the project information of a software environment, the integration of a query language into a software environment to answer unanticipated queries for the user and to aid in tool integration.

5. Other interests: operating systems, programming language theory, compiler theory, and software engineering.
Publications


# Table of Contents

Acknowledgement ........................................ iii
Vita .................................................................. v
Table of Contents .......................................... ix
List of Figures .............................................. xii

1. Introduction .............................................. 1
   1.1. Overcoming Entropy .................................. 1
   1.2. Major Thesis .......................................... 6
      1.2.1. Efficient Integration of Existing Tools ....... 7
      1.2.2. Effective Integration of Existing Tools ..... 8
      1.2.3. Ergonomic Integration of Existing Tools .... 8
   1.3. Contributions of this Research .................... 9
      1.3.1. Parameters of Tool Integration ............... 9
      1.3.2. Concepts and Techniques Useful in Integration 10
      1.3.3. Impact of this Research ...................... 13
      1.3.4. Limitations ...................................... 14

2. Tool Integration: The Problem and a Solution ......... 16
   2.1. The Tool Integration Problem .................... 16
   2.2. Benefits of Tool Integration ..................... 25
   2.3. Integration of Existing Tools .................... 26
   2.4. Method for the Solution .......................... 28
   2.5. Environment for the Solution .................... 31

3. Related Work: Recent Tools with Radically Different Architectures .......... 36
   3.1. Operating Systems and Concepts .................. 37
   3.2. Programming Environments ....................... 42
      3.2.1. Programming Environments Based on a Structure Editor 43
      3.2.2. Programming Environments which Supports the Life Cycle of Program Development 49
      3.2.3. Artificial Intelligence Concepts and Systems 54
   3.3. Other Work on Tool Integration .................. 58
4. The Parameters for, and a Categorization of, the Degree of Tool Integration

4.1. Components of the System
4.1.1. User Interface Design
4.1.2. Software Tools
4.1.3. Information Base

4.2. Parameters of Tool Integration

4.3. Categorization of Tools for Integration
4.3.1. Granularity
4.3.2. Cohesion
4.3.3. Harmony

4.4. Categorization of Integration - Loose versus Tight

4.5. Analysis of Some Typical Existing Systems in Terms of this Categorization
4.5.1. Unix Pipes
4.5.2. The Cornell Program Synthesizer
4.5.3. Pecan

4.6. Application of these Concepts to Other Domains
4.6.1. Important Features of the Domain of Software Project Development
4.6.2. The Domain of VLSI Design
4.6.3. The Domain of Business Data Programming

5. Integration of Existing Tools

5.1. Environmental Requirements for Integration of Existing Tools
5.1.1. Requisite Properties of the Operating System
5.1.2. Requisite Properties of the Information Repository
5.1.3. Requisite Properties of the Tools

5.2. Useful Concepts/Constructs in Achieving this Integration
5.2.1. The Monitor
5.2.2. A Method
5.2.3. The View Extractor and Output Distributor
5.2.4. The Model
5.2.5. The Project Information Repository

5.3. Useful Techniques for Integration of Existing Tools

5.4. Advantages of this Approach to Integration

5.5. The Human-Engineering Potential of Harnessing Existing Tools
6. Implementation

6.1. Design of the Forms 114

6.2. View Extractor for the C Compiler 118
  6.2.1. Global Control for the View Extractor 118
  6.2.2. Local Action Routines for the View Extractor 118
  6.2.3. Attributes Use by the View Extractor 123

6.3. Output Distributor for the C Compiler 124

6.4. Debugger Integration 124

6.5. Commands 126
  6.5.1. Primitive Commands 126
  6.5.2. Composite Commands 131

7. Summary and Conclusions 134

7.1. Summary 134
  7.1.1. Motivation: The Advantages of the Integration of Existing Tools 134
  7.1.2. Characterization of Tool Integration 136
  7.1.3. Useful Techniques 138
  7.1.4. Demonstration of Feasibility 140

7.2. Conclusions - What Has Been Learned 140
  7.2.1. Loose Levels of Integration 141
  7.2.2. Integration of Existing Tools 142

7.3. Future Research Directions 143
  7.3.1. Implementation Extensions 143
  7.3.2. More Conceptual and Theoretical Extensions 145

Bibliography 147
List of Figures

Figure 1: Some Important Terminology 4
Figure 2: Some Important Terminology (continued form the previous figure). 5
Figure 3: Un-Integrated Tools Lack Cooperation 18
Figure 4: Conversions between Each Pair of Tools in an Un-Integrated System 21
Figure 5: Communication and Cooperation through Tool Integration 24
Figure 6: Forms and Underlying Productions 29
Figure 7: An Informal Method for Evaluating a Tool for its Integration Potential - Granularity and Cohesion of the Information Repository and the Tools 76
Figure 8: An Informal Method for Evaluating a Tool for its Integration Potential - Granularity and Cohesion of the Monitor 77
Figure 9: An Informal Method for Evaluating a Tool for its Integration Potential - Harmony versus Dissonance 78
Figure 10: Evaluation of the Integrability of Some Existing Systems and Components 81
Figure 11: Comparisons of Some Measures of Effort 103
Figure 12: Forms Containing a C Program 117
Figure 13: The Global Control for the View Extractor 119
Figure 14: The Errors Inserted in the Forms by the Output Distributor 120
Figure 15: Some of the Local Action Routines for the View Extractor 121
Figure 16: The Output File Produced 122
Chapter 1
Introduction

1.1. Overcoming Entropy

Entropy is the general trend toward disorganization as depicted in the second law of thermodynamics: "When energy is transformed from one form to another, some energy is always converted from a usable form to an unusable one". The origin and primary realm of application of this law is in the field of the physical sciences. Extensions into other orbits, as indicated by Rifkin [53], are not without reality. Rifkin's application of the principle of entropy to ecology and to societal interactions produces a melancholy world view that has more truth than we care to admit.

Entropy, that is the general trend toward disorganization, is one of the driving forces behind the need for computer science. Entropy is the prime source of complexity in our world in general, and in information in particular. Contemporary computer science is much more than the study of programming, algorithm development, information handling, or any other similar definition [42]. As a whole, computer science is a confluence of ideas focusing on the study of complexity and methods of controlling that complexity. Succinctly, computer science is the study of methods of
overcoming entropy. In a retrospection of the early history of computer science, it is evident that one of the major conceptual achievements was the realization that the computer could be used to manipulate the very data that was used to control it. Von Neumann's stored program concept lead to the development of assemblers, then to other compilers and translators. The die was cast. Computer software could function as a tool to help a human being optimize his/her efforts by performing in a more reliable and efficient manner those tasks which can be automated.

Immediately, there began a tendency toward disorganization in these software tools and in the information which they manipulate. Each tool was designed to optimize its input and output requirements to simplify its own task with little or no regard for the manner in which its input is produced or its output used. For example, compilers often require that the source program reside in a single file. (This problem has been solved in part by separate compilation of modules and more incremental compilation.) The results of a compiler in the case that syntax errors exist has been a list of line numbers and cryptic error messages which are difficult for the user to understand and correlate to correct the program. Text editors produce text files even when used to construct

\footnote{An area of computer science of special interest is software engineering and development. Project development generally begins quite informally as the discussion of solutions to a problem. Management tools usually fail to record these discussions, the early decisions made because of them, or the motivations for the decisions. Attempts are made at later stages to record requirements, specifications, and design decisions. But entropy has often already begun its destructive work. The complexity of the system has quickly reached proportions that make it unwieldy, if not impossible, to manage. Software development methods, by structuring the solution, can begin to overcome this entropy by reducing the complexity. The TRIAD software environment partially automates the application of such a method.}
programs. The structure of the program is ignored even though obvious in the text produced. (Structure editors have improved this situation immensely.) Furthermore, each tool presented a different interface to the user which was designed to satisfy its own requirements, with no concern for uniformity or consistency.

The entropy of the information is being overcome with the realization that all project information should be retained in an organized manner, and with the application of database technology to this domain. The current emphasis on user friendly software (see Figures 1 and 2) has given impetus to the improvement and the consistency in the user interface presented by tools of a particular system. Surmounting the entropy of tools is the task of tool integration. To perform optimally, tools should be designed to cooperate and communicate with the ultimate aim of enabling the user to better accomplish his/her goals. Each tool may, as a result, not be precisely "tuned" to optimally perform its own task. But the synergistic effect of this tool cooperation is its own justification.

Most recent work in the research and development of software environments that recognizes the need to provide better tool integration and communication concentrates on the use of radically different architectures and new tools that are cognizant of, and communicate with, each other. We provide an alternative architecture by which existing compilers and other tools can be retrofitted into a "harness", consisting of an interface and a project information base, based on an extended attributed grammar model. One advantage of this is in the ergonomic benefits of having an incremental, project-oriented interface. Specifically, the harness facilitates commands which support the software
DEFINITIONS

• **Tool Integration** - a cooperation and communication among the software development tools and the project information repository which permits increased tool effectiveness and user friendliness.

• **User Friendliness (ergonomics)** - the quality possessed by a software system or component (e.g. a tool) which is typified by the following characteristics:

1. Tool results are presented at the user's focus of attention.

2. Tools are invoked from the user's current focus (i.e. there are few tool transitions of which the user is aware).

3. Tools are addressed by a uniform command syntax which includes a concern for efficiency of keystrokes.

4. Tools provide rapid user feedback because of their interactive and incremental nature.

5. Tool modes of operation are eliminated and tool transitions are hidden from the user.

*Figure 1:* Some Important Terminology

engineering process in ways not possible with existing, isolated tools and compilers. The tools can actually be increased in power by providing
DEFINITIONS
(continued)

- **Software Environment** - a software system to support the software development process through all its phases. Such an environment should permit the storage and retrieval of all project information in an easy manner. This term will be used to refer to environments which possess most of the following properties to some degree:

1. interactive with the user,

2. incremental operation of the tools,

3. user friendliness (as defined above),

4. syntax directed editing, and

5. extensible.

**Figure 2**: Some Important Terminology
(continued from the previous figure)

more powerful commands. A further advantage in the retrofitting approach is in the economic benefits of retaining the use of existing tools which are now fully robust and have been used to develop large volumes of operating software.
1.2. Major Thesis

The major thesis of this research is that the method-driven integration of existing tools into a software environment can produce a more efficient, effective, and ergonomic environment. (See the definitions in Figures 1 and 2.) Furthermore, this method-driven integration has been demonstrated to be feasible in a test bed in which it was quite economic. More precisely, the pragmatic goal of this research is to permit the conversion of useful but out-moded software tools to the state-of-the-art in user-friendly software environments in which the tools are integrated, interactive, and incremental. (Refer to figures 1 and 2 for a more precise definition of those terms as they are used in this dissertation.) Thus the baseline is an unintegrated set of traditional tools and a software environment containing a monitor/user interface and an information base for all project information (e.g. the TRIAD environment). The goal is an environment which is at least as powerful and as user friendly as other state-of-the-art software environments (e.g. Cornell Program Synthesizer [66], Gandalf [17], Magpie [6], Pecan [49], etc.). The thesis is that this can be achieved, in the harness provided by a TRIAD-like environment, in an economic manner which is much less expensive than the effort necessary to develop new tools to accomplish these tasks. The conclusion

---

2The adjective method-driven is used to modify the term integration to distinguish this work from other uses of that term. In other research [31, 32, 2, 3], integration pertains to the process of providing techniques for allowing software tools from heterogeneous environments to be used at remote sites. The prime objective is to allow these tools to be used without the user having to be concerned about the location of various global objects (i.e. tools and files). The aim of the research described in this dissertation is to enhance the support provided by existing tools in a given environment by achieving a level of communication and sharing among the tools and other system components which is usually not present. The term integration will be used to refer to this method-driven integration.
is that resources expended in the development of tools which are not significant improvements in a conceptual sense are wasted. They would be better directed at the integration of existing tools into an existing environment.

1.2.1. Efficient Integration of Existing Tools

The cost of developing and maintaining software is the major expense in the budget of most data processing departments. This expense is no less for the development of robust tools. The long term gain in the increase in productivity invariably offsets the initial outlay in company resources. Given the investment that most organization have in such tools, good stewardship demands that these be well used as long as is profitable. The integration of these existing tools in an environment which increases their power and usability can prolong their life and, thereby, protect a company's investment.

This investment of resources is not limited to those of the computer itself, but extends to the time and effort expended by the users of the tool in learning the characteristics and idiosyncracies of that utility. Given the value of the time of the programmer or user, the protection and optimal use of this human resource is more critical than that of the computer system's resources. The users' knowledge of, and confidence in, existing tools is an extremely valuable company resource which should not be overlooked.
1.2.2. Effective Integration of Existing Tools

As a consequence of the cooperation and communication that results from this integration of tools, the tools can work together to accomplish that which none could by itself perform. The results of one tool can be used to modify the input of a second tool. Information obtained from the information repository can be used to modify the commands to a tool. The synergism that results effectuates a more powerful set of tools. Consider the integration of the monitor or database query mechanism with a version control system. Such a combination of tools could produce the answer to managerial question like "What versions of the system being developed depend upon modules written by programmers in John Doe's group?". Spelling checking programs have become much more usable when coordinated with an editor to allow for easy correction. This (method-driven) integration can enhance existing tools to the point of providing commands that are more powerful than are possible with more sophisticated tools.

1.2.3. Ergonomic Integration of Existing Tools

One outcome of the integration of tools is the unification of the user interface to these tools. The ergonomic advantages of a common interface are clear. The user is unaware of the application of tools, and can focus his/her attention upon the project information. The idiosyncrasies of each tool are minimized and a commonality is established.

The results of the tools in such an integrated system can be presented in the appropriate context in the project information. For example error messages from a word processing system (like Scribe [48] or Runoff [11]) would be more easily understood if they appeared in the text. The
meaning of data in context is much easier to determine. Again the user is able to maintain a level of concentration on the project information rather than having to continually switch contexts.

1.3. Contributions of this Research

The primary contributions of this work in the integration of existing tools into a software environment are:

- a categorization of parameters for, and levels of, tool integration,
- techniques for the integration of existing tools into a software environment, and
- a demonstration of the feasibility of such an integration.

1.3.1. Parameters of Tool Integration

The integration of tools does not represent a precisely defined state of being or quality of interaction, but rather a continuum of degrees of communication and cooperation. (The extremes of the continuum are loosely categorized as loose and tight integration.) The parameters that characterize these levels are:

- Granularity,
- Cohesion, and
- Harmony

These parameters, to be discussed later in great detail, have a variety of
meanings when applied to the various components of a software environment. Taken together, they produce a taxonomy of levels of integration that is useful to consider in determining the relative advantages and costs of various techniques and levels of integration.

1.3.2. Concepts and Techniques Useful in Integration

Having categorized and parameterized the levels of tool integration, we proceed to an exposition of various constructs, concepts, and techniques that make an integration of existing tools reasonable and pragmatic.

To become integrated into the software environment, an existing tool must obtain its project information from, and store its results in, the canonical information base. This requires a "view extractor" and an "output distributor". The view extractor for a particular tool collects the necessary information from the project information base and presents it in the form necessary for the tool to use. To accomplish this the view extractor needs a knowledge of both the structure of the project information base and the format of the tool's input. The output distributor places the results of the tool back into the appropriate context of the project information. This may require some supplementary information from the view extractor that produced the information. As should be apparent from the discussion of the commands, the architecture for integrating existing tools should thus include the following components:

- **View Extractor**
  - for presenting human engineered views of the information base and the output results of tool applications, and
for presenting the view needed by a particular existing tool.

- **Output Distributor**
  - for inserting results into the structural context in the canonical information base.

- **Monitor**
  - for maintaining global contextual information, and
  - for coordinating and controlling all the tools.

- **Attributes**
  - for retaining tool status information, and
  - for storing local information.

- **Action Routines**
  - for maintaining accurate attribute values,
  - for automating instrumentation, and
  - for propagating attribute values.

The climax of this research was the illustration of the practicality and feasibility of these techniques and constructs by the integration of some existing tools into a software environment. The tools chosen for integration, a C compiler and a symbolic debugger, were used without
internal modification. Their integration produced a system that gave the user feedback and control resembling that of incremental tools. The number, variety, and power of commands to each tool individually were increased. Furthermore, the user's interface to the tools was simplified and made more uniform. The human engineering of the user interface is widely recognized as an important aim of a computer system. Discussion of this topic generally produces few specific recommendations. This implementation incorporates some ideas and techniques which add in a substantive way to the user friendliness of the interface. Initial use of the system seems to verify the utility of these ideas.

The test bed demonstrates the practicality of the integration of existing tools. The major development efforts centered on the creation of a set of forms for the C programming language [23], the coding of a "view extractor" and "output distributor" for the C compiler, the integration of the DBX debugger, and the development of a group of commands for the integrated set of tools. (These will be discussed in great detail in subsequent chapters.) The effort expended, while significant, pails in comparison to that spent in the coding phase of compiler construction, not to mention the effort necessary for the requirements and design phases. (See figure 12.) Thus, the advantages of integrating existing tools (increased tool power, synergism of tool cooperation, human engineering of the tool interface, etc.) can be gained at a relatively small cost.
1.3.3. Impact of this Research

The primary impact of this work lies in three areas:

- economic benefits,
- ergonomic advantages, and
- increased effectiveness.

The economic advantages arise from the relative cost of integrating an existing tool and of constructing a new version of a similar, sophisticated tool. As discussed above, the test bed illustrates some techniques that can simply and efficiently produce such an integration.

The user friendliness of the integration techniques springs from the exploitation of the monitor (alias user interface). This monitor provides the user with a common and consistent interface to the tools. By intervening between the tools and the user, the monitor can modify user commands and can inbed the tool results in the correct syntax. The modification of user commands enables the use of a better designed, more consistent command syntax. The syntax required by the tool is not a constraint since the monitor can pad or modify the command to make it acceptable to the tool. Conversely, the monitor's inbedding of tool results in their appropriate context conveys much more information to the user.

The integration concepts and techniques described in this research add to the effectiveness of the system by increasing the power of commands to each tool and by permitting new commands which exploit the synergism of tools in cooperation.
1.3.4. Limitations

To be comprehensive, the limitations of the applicability of these techniques and concepts for the integration must be presented. The domain of applicability is somewhat restricted by requirements on the supporting operating system. The primary operating system necessity is the possession of a subprocess capability. The integration of existing tools under the control of an centralized monitor mandates that these tools execute as subprocesses of this monitor. An operating system which does not supply this subprocessing ability or an equivalent mechanism is not an acceptable foundation for the application of the results of this research.

The tools have some constraints in order to be candidates for this integration. Such a tool must permit its input data to come from an unaccustomed source, the monitor. Further, the tool must permit its output to be dispersed to alternative locations. (An appropriate operating system may allow these capabilities in a manner which is completely shielded from the tools.) A tool without these properties is not an adequate candidate for integration. There are other properties that, although not requisite for integration, enable a "tighter" level of integration and a resultant higher degree of user friendliness. These properties include that of having a substantial level of interaction with the user. Even greater improvements can be accomplished by means of incremental tool operation. The presence of these properties is not crucial to the ability to integrate a given tool, but are critical in determining the level of integration possible.

Another limitation is that, in some situations, the integration, while possible, may not be feasible or practical. A lack of incrementality may
mean the response time to the user is too long to be acceptable. This is a limitation of practicality rather than applicability. An analysis of the integrated tool's responsiveness to the user should proceed any extensive application of these techniques.

An examination of the above limitations will indicate that they are not especially restrictive. The techniques and concepts of this research would seem to have a wide range of applicability.
Chapter 2
Tool Integration: The Problem and a Solution

2.1. The Tool Integration Problem

In describing the nature of the tool integration problem, it is crucial to agree upon the terminology used. A tool, in the context of software development, is a software component which aids the user in the performance of his/her task. This assistance may vary from actually completing these tasks that can be automated, to giving advice or providing data. Tools fall into the following general categories:

- **Editors** - Text and structure editors have the labor of enabling the user to change the contents of a buffer or file. A text editor performs its activity on a file or buffer of characters. A structure editor has an embedded knowledge of the syntax of a grammar underlying the set of characters.

- **Translators** - Compilers and interpreters of myriad types have the endeavor of translating from one syntactic form to another while retaining the semantics. The development of this concept of automated translation from one (usually higher level) language to another (lower level) one was one of the major accomplishments of the early history of computer science. This concept continues to evolve as illustrated by current research on incremental compilers [6, 49, 13].
• **Project Management Aids** - Systems and notations which provide management support to monitor and control project development include version control systems [68], source code control systems [55], cost estimators, and report generators.

• **Analysis Tools** - This type of software examines the project information to find patterns of usage that reflect some meaningful information to a manager, an系统 analyst, a programmer, or a user.

• **Formatters** - Report generators and pretty printers [33] present portions of the project information to the user in a form which is more easily consumed and digested.

This taxonomy of tools may not be exhaustive now and certainly will not be in the future. This is a fertile area of computer science research. Areas such as compiler technology (in incremental compilation), artificial intelligence (in expert systems), database systems, distributed systems, and software engineering collaborate in the evolution of new, more functional, and more ergonomic tools.

Software utensils of this type have proven their usefulness and even their indispensability to software development over the past thirty years. Computer and human resources are more effectively and efficiently used. The user can concentrate on the more creative aspects of project development. By specifying the necessary processing in an algorithm implemented in a tool, many tasks are performed with increased accuracy and reliability. Furthermore, these tasks can be completed much faster than by human effort. Consequently, the effective work accomplished by the confluence of the efforts of a user and software tools is increased several orders of magnitude.
The primary conceptual limitation of most software development tools as they currently exist is a lack of cooperation among these tools. See figure 3. This lack of collaboration often effectuates a duplication of effort as tools performing different, but related, tasks share the need for some of the steps in the tasks. For example, an accounting package may produce a file of records, each of which consists of various fields. If this output is to be later sorted on one of these fields by another tool, this sort tool will have to reparse the file to recover the fields of which the first tool had full knowledge. This duplication can also occur within a single tool from one application of the tool to the next. A compiler is used repeatedly in the development of a source program which slowly changes. Most existing compilers completely retranslate the program despite the fact that symbol table entries, tokens parsed, and even sections of code produced are identical from one invocation of the compiler to the next.

![Figure 3: Un-Integrated Tools Lack Cooperation](image-url)
To be more specific, the lack of cooperation and communication is frequently a result of the preoccupation of many tool designers with the optimization of each tool's input and output format in ways that will facilitate its own job. The input to one tool which arises as output from another requires a translator to produce compatibility. If tools were instead developed with the optimization of the entire software development process as a goal, the user would ultimately be better served. This would inevitably require some tools to perform added processing to conform to the standards of the system although the task of most components would more more easily discharged. One such standard that would often prove pragmatic is a common information representation for use by all tools in their input and output. (The Unix system has achieved a certain level of tool cooperation by establishing a standard for tool communication - the character stream.)

Another aspect of this lack of cooperation among tools is the absence of a common user interface. Each tool has its own syntax, conventions, and modes of operation. The user, particularly the novice, is faced with a plethora of incongruent interfaces. The tasks of various tools diverge sufficiently that some variation in interfaces is necessary. But much more commonality could be achieved, especially for the set of tools to be used in a particular environment.

More specifically, the (interrelated) problems with lack of tool cooperation and communication are:

- The dissonance in the information (software-related project information), that is used and produced by various related tools, causes valuable data to be not communicated between the tools, and, thus, irretrievably lost. This is because tools do
not share a standard organized method of storing and communicating information to other tools that could profitably use it. For example, even though a typical compiler produces a parse tree data structure from the source file, a pretty printer tool \[33\] to format the output will very likely reconstruct the structure of the program from the source listing. This leads to a lot of redundancy in data, problems in maintaining consistency between the different data structures, and duplication of processing. Conversion packages between the different data structures may be written. However, this involves writing a conversion package for each pair of tools and the programming effort involved is considerable. See figure 4. Consequently, tools, that ought to communicate via shared information, often never do.

- The level of granularity of the information which the tools manipulate is often inappropriate for effectively communicating results to the human user. In most existing systems, tools work either at the file directory level or at the text level. Consequently, tools typically process the complete file before they provide any feedback to the user.

- Even if the granularity of the information is very flexible, the lack of structure among the grains of project information, can obviate any advantages of the fine granularity. For example, a text editor tool uses information at the character level of granularity, but has no knowledge of the structure of that information. The tool which produced the information was aware of the structure, but had no standard method of
recording this structure and communicating it to the tool requiring this information.
The terms "granularity" and "dissonance" will be more precisely defined and more completely discussed in a subsequent chapter.

The above problems suggest the use of a standard, structured information repository to be used by the tools in the system to store all project information. Additional problems due to the lack of a tool-kit interface for these tools must also be considered:

- In industrial projects, the volume of project information (consisting of design documents, code, and management plans) is enormous. Due to human limitations, it is therefore important for the project members to avoid shifting their attention from one chunk (or conceptually related grouping) of information to another. All the tasks (editing, debugging, analysis, etc.) should be supported at the user's focus of attention. That is, the programmer should be able to concentrate on a chunk, rather than on the tools that are used to accomplish the tasks, and be able to obtain immediate feedback, incrementally, at that focus of attention.

- There is no support for the "process" that the user must go through to design and manufacture the software. That is, there is no support for the methods that structure the process by which software is created. Typical methods, like Structured Design [63] or Jackson [20], incorporate patterns of programming or management which have been developed through experience.

- The user's focus is not on the information, that is on its creation and use, but on tools which are actually secondary to
the user's interests. Further, there is no uniformity in the interface presented to the user. To use each tool, the user must operate within a unique "mode" and memorize the quirks of that mode of the tool.

- Tools tend to be monolithic. That is, to be a viable commodity in today's marketplace, each tool has to provide a comprehensive set of functions in order to be usable by a wide variety of groups. Rather than avoiding duplication by delegating functions to other tools, each tool is built to be comprehensive and, consequently, is quite complex, difficult to build, a horror to maintain, and virtually impossible to adapt to changing needs.

- Tools must often contend with complex interactions between global and local information. To perform some processing that is localized to a small piece of software-related information, the tool must often store or recover global contextual information. This suggests the need for an modular architecture based on a clean model which separates the global tool logic from local, chunk-related details.

- New tools are often not able to work with existing tools. That is, new tools cannot readily use information created by the old tools.

The interface-related problems suggest the need for an interface monitor which supports appropriate methods during the life-cycle of the product, as well as provides a uniform, "intelligent" interface to a variety of tools.
Therefore, software development tools can cooperate more potently by cooperating and communicating. See figure 5. This implies a level of tool integration. Tool integration is defined here to mean the cooperative operation of several tools to advance the overall goals of the project. This integration can be characterized by:

- cooperation - sharing of control.
- communication - sharing of information, and
- commonality - sharing of interface.

Cooperation is epitomized by incremental tool cooperation. A tool which
operates incrementally is able to use results from previous applications of the tool to avoid replicating work. Furthermore, each invocation of the tool results in the tool completing a small segment of the task that must be repeated or accomplished for the first time because of changes in the source information. Cooperation among such tools can easily occur as control flows among the tools and the user in a co-routine manner.

Communication among tools implies, as mention previously, the use of a common format for the information. Such a standardization may require a change to a tool or else a translator to convert to the standard form. Either solution produces tools with the potential for communication by means of this common information representation.

A common interface to the user is a necessity in order for a set of tools to effectively communicate with the user. This again imposes the need to either change the tools or to develop an outside agent capable of producing this uniform syntax for the user.

In summation, tool integration can conquer the communication and cooperation vacuity inherent in most sets of tools. In addition to these negative motivations for the integration of tools, some compelling advantages of a more positive nature are recounted.

2.2. Benefits of Tool Integration

The conceptual benefits of tool integration are dual. First tools in cooperation have a synergistic effect which is is greater than any single tool could achieve alone. By evading much of the duplication of effort (as in repeated parsing and unparsing of data), cooperative tools are able to achieve a greater efficiency. (In the contemporary time of relatively
inexpensive hardware, efficiency would seem to be unimportant, except as it influences the response time to the user. The optimization of the user's time is a worthy goal.) In addition to the added efficiency, combination commands, which are more powerful than commands to the tools individually, can be developed. (Examples of such cooperation and the possible commands will be enumerated in later chapters.)

The second conceptual advantage of integrated tools is an increased level of user friendliness. This is notably demonstrated in the common user interface. By presenting a uniform syntax to the user for all tools and a consistent interface within a tool, idiosyncratic differences among tools and pernicious modes within a tool can be avoided. The denouement is the user's concentration of his/her attention upon the project information. Ultimately the user is no longer aware of the tools, but can focus on the project information.

2.3. Integration of Existing Tools

Having established the advantages of the integration of tools, we now proceed to consideration of the integration of existing tools into a software environment. Such an integration has, of course, all the advantages of integrated tools, i.e. more powerful commands and increased user friendliness of these tools. The use of existing tools in such a cooperative environment has additional advantages, not the least of which are economic considerations. Creating, i.e. designing and coding, a new software tool can be an expensive operation in terms of computer and human resources. The cost of integration of existing tools lies in providing the appropriate view of the information base to the tool and placing the results of the tool back into the correct context of the information base. Both of these can be accomplished relatively
inexpensively as will be demonstrated by means of an implementation. (This integration of an existing compiler and a symbolic debugger into a software environment will be discussed in the chapter on implementation.)

A more vital issue is the level of confidence in a tool that a user develops after a long record of dependable service. Newly produced tools have not been proven reliable, and often evolve through several versions in which improvements are made and serious errors corrected. Even though a new tool is a technological advancement, it may not be initially perceived as such by the user. Furthermore, the user's productivity will normally take a temporary nose dive as he/she learns the idiosyncracies of the new tool which have long been mastered in an older tool. A sophisticated tool often has a detailed set of commands and capabilities that are best learned through experience.

The past decade has witnessed a great amount of work in the research and development of software tools. Contemporary tools are quickly outdated by this progression. Existing tools are expanded in power and increased in usability. Occasionally a technologically advanced tool, with no closely related ancestor among previous tools, will produce a quantum leap in our understanding of tool capabilities. A software environment which is capable of integrating tools as they develop and prove their reliability can sustain its utility. Most environments include a great variety of tools. If these tools can be replaced on an individual basis while retaining the remaining tools, the integrity of the environment is maintained. Such an evolution of the set of tools in the environment's tool kit reaffirms the users' faith in the reliability of the system. An environment which can incorporate existing tools has increased flexibility
2.4. Method for the Solution

To demonstrate the feasibility of an integration of existing tools, and to verify the increased power and usability of such an integration, a test bed solution was implemented. The TRIAD software environment [26, 27, 28, 46, 59] was chosen as the foundation upon which the integration was to occur. This environment provides an information base in the form of a tree of project information partitioned into forms, and it furnishes a monitor or user interface. These forms are a view of the attributed grammar which underlies the project information. See Figure 6. Since this interface is form-driven, C programming language forms were developed to include program code in the information base (Figure 13). The existing tools chosen for the integration were a C compiler and DBX [5], a symbolic debugger. These were integrated with the information base. This integration was accomplished with no changes to these tools.

The C compiler is integrated by the following general strategy. (The details will be discussed in the chapter on implementation.) C programming language code resides in the programming forms in the information repository. This code is extracted into a file to produce a correctly ordered, syntactically complete C program. The C compiler is invoked as a child process with this file as its input source. If the compilation produces errors, each error statement is inserted next to the
### General Loop Construct

<table>
<thead>
<tr>
<th></th>
<th>Change History</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>prompts</strong></td>
<td><strong>Change Date:</strong> (mm/dd/yy)</td>
</tr>
<tr>
<td><strong>WHILE</strong> (condition) <strong>DO</strong></td>
<td><strong>Change Item</strong> <strong>Change Description:</strong></td>
</tr>
<tr>
<td><strong>BEGIN</strong></td>
<td><strong>Rationale:</strong> Due to change in specification of sender</td>
</tr>
<tr>
<td></td>
<td><strong>Possible Impact:</strong></td>
</tr>
<tr>
<td><strong>X := Y</strong></td>
<td>Use X instead of Y</td>
</tr>
<tr>
<td><strong>(statement)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>(statement)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>END</strong></td>
<td></td>
</tr>
</tbody>
</table>

**chunk of project information**

**A. Example of Forms**

1

\[
\text{general loop construct'} := \text{WHILE (condition) DO (statement') END}
\]

2

\[
\text{change history'} := \text{change item'} \text{change item'} \text{change history'}
\]

\[
\text{change item'} := \text{change item'} \text{change history'}
\]

\[
\text{date'} := \text{description'}
\]

\[
\text{rational'} := \text{possible impact'}
\]

### B. Underlying Productions

**Figure 6:** Forms and Underlying Productions
offending statement in its original form in the information base. See Figures 14 and 15.

The DBX debugger is also invoked as a child process of the monitor. As it continues to run in the background, the user is permitted to traverse the database, invoke other tools and, when desired, issue commands to the debugger. The nature of these commands is much improved over the original DBX commands. Their syntax is adapted to better suit a friendly user interaction. Each is mapped to the corresponding command or set of commands to DBX. For example, to set a breakpoint at a particular statement of the program, the user positions the cursor on that statement and issues a simple command. The monitor determines the line number of that statement and uses this data to pad the command syntax that is required by DBX.

The implementation chapter describe a complete set of commands that has been developed for the integration of these tools. One subset of commands results in the compilation of various combinations of the code included in the project information. Not only can the entire program be compiled, but the syntax of a single procedure can be tested and procedure interfaces can be checked for consistency. The debugging commands include the setting of breakpoints at given statements, before, during, and after a control structure, or in a procedure. The value of a particular variable can be printed or changed. A section of code can be instrumented to monitor its operation. Combination commands can involve retrieval of information from elsewhere in the project information which can be used to modify the behavior of commands. The accumulation of all these commands with all their possible interactions is a set which can enable a user to quickly develop and debug C language programs stored in the project information base.
2.5. Environment for the Solution

The TRIAD software environment forms the foundation or test bed upon which the feasibility, economy, and power of the solution (i.e. the integration of existing tools) is demonstrated. (The solution is not limited in its application to TRIAD alone. The ideas and techniques have relevance for most, if not all, software environments.) To understand the current implementation, a more detailed knowledge of TRIAD is necessary. Several important concepts form the theoretical basis for this environment. The most important one is the grammar which molds the information base. (This model is actually an attributed grammar form [34, 1].) This grammar provides the structural information to relate the various portions of the project information. The model permits the user and the tools to retrieve and analyze the data generated during project development. More importantly, this grammar form model encodes a software development method that structures and standardizes the development of the project.

This grammar is presented to the user as a method-driven forms interface. The process of software development appears to the user as the task of choosing and completing (that is, filling in) the appropriate forms. The user's choices at each step are structured and limited to the productions of the grammar that can validly be applied at that point. The headings (or organizers) of the forms represent the symbols of the grammar. The process of applying a production to a nonterminal is represented by refining a heading of the form, i.e. a nonterminal, with another form. See Figure 6. The project information that is produced in this process is thereby standardized and structured.

The use of forms has the additional benefit of reducing the amount of
memorization of arbitrary syntax that is required. The resultant user interface furnishes a desirable, easy-to-use system to the user while helping to enforce the standards and the method required by those managing the project.

This grammar can be extended to cover all components of a system, i.e. management details, design documents, requirements specifications, and programming language code. By including the grammar of a programming language as a portion of the method's grammar, a program editor (in the sense of Poe [13], ALOE [36, 35], and others) is developed.

The presence of attributes in the attributed grammar form is a useful mechanism. These attributes are descriptive adjectives attached to particular logical components or chunks of project information. The result of allowing such descriptive data to be stored in its context is a more efficient representation of that information. Some of the semantics of the data is lost when it is taken out of context. (This contextual information could be replicated with the attribute in some other location, but the space wasted, the added time for retrieval, and the consistency problems make this unattractive.)

A semantic function can be specified to be associated with each attribute. This function specifies the processing necessary to keep the value of the attribute current and accurate when changes occur in its context. This procedural component can also specify how a change in the contents of this attribute should affect similar attribute in nearby portions of the information base.

The existing TRIAD software environment consists of several major components and subsystems. The monitor, or user interface, provides
access to the project information by means of a myriad of operations. Some of these commands allow the traversal of the information base in various manners, e.g. moving to the next or the previous heading, to the next or previous form, to the first form, etc. Others perform dynamic searches by searching for a given string over all the headings or in the entries of the forms. More sophisticated commands provide various summary reports by collecting the appropriate information from the base. Commands are provided to insert new instantiations of forms into the database (i.e. to refine headings with other forms). The monitor is also the component that supplies the interface to all tools.

The TRIAD information base is tree structured in that the forms are related hierarchically. (The traversal commands mentioned above use this structure to provide functions that move to a child form or to the parent form.) This structure seems to best reflect the dominate relationships in the management information, design requirements, and program code. Many software development methods fit well into such a tree structure for the information produced.

The grammar that describes the method used to drive the system and to structure the information base is not intricately woven into the fabric of the TRIAD system. More than one method can be supported. A database of existing methods exists, each of which can be used to drive the system. Software development methodology has not reached the point at which any one method can be declared to be optimal for all situations. Each company or organization sets its own methods when it establishes standards. By allowing the method underlying TRIAD to be changed, flexibility is gained. TRIAD can be adapted to any standards or method by the design of a grammar to describe the method.
These methods and standards often evolve within a given company. New requirements are added, others are relaxed, as experience teaches management what works best. The TRIAD tuner is a system utility provided to adapt, or "tune", a grammar to reflect these changing standards and methods. The static tuner, which is itself form-driven through "meta-forms", allows a new method to be developed by the specification of the productions, terminal and nonterminal symbols, the attributes and semantic functions, and display information. The dynamic tuner [34] allows forms and trees of forms that were designed under a previous method to be changed to work with a new grammar. (There obviously must be some restrictions on the degree to which the two grammars differ.) This allows portions of prior projects to be re-used in newer projects despite the fact that the grammar describing the method has evolved. One particular item of interest to this research on the integration of existing tools is a set of forms describing the C programming language. These forms (see Figure 13) allow C Programs to be develop in the TRIAD environment. The use of forms to display the programming constructs, as it currently exists, is cumbersome and not optimal for the user. (The forms interface excels at supporting other phases of project development.) But the power provided by the underlying grammar makes this acceptable, especially for a research instrument.

The current TRIAD environment also furnishes the capability of specifying action routines to be associated with local pieces of information. (They are actually associated with symbols of the grammar.) These action routines are quite useful for specifying local procedural actions to be performed when something happens to the chunk of information to which they are attached. In particular, a separate action
may be specified to be invoked or fired on entry to, exist from, creation of, or modification of a chunk of information (actually a symbol of the grammar);

This is the environment upon which the implementation described in this dissertation is based. The facilities provided proved to be more than sufficient. This particular environment entails some features that make the task of integrating existing tools quite simple. The techniques described are not limited to this environment but have applicability to most current software environments.
Chapter 3
Related Work: Recent Tools with Radically Different Architectures

Some of the newer tools and environments are based on radically different architectures. These approaches have resulted in tools that, from the perspective of the user, are substantial improvements. One such architecture is the object-based system such as Smalltalk [16]. Using a more distributed control concept as the architecture, this type of system is able to readily support a tool-kit architecture which facilitates the addition of new tools and tool fragments to complete a task.

An opposite approach has also proved to be a successful paradigm for improving productivity while developing small programs. This is the tight integration of tools [21] and it is typified by systems like the Cornell Program Synthesizer (CPS) [66, 65, 51] and Gandalf [17]. Rather than distributing the control, the tools are tied together by their use of a common data structure. In CPS, the editor and interpreter both store and retrieve their data from a parse tree. The resulting close cooperation between tools enables the user to concentrate on the development of program-related information, rather than on the tools. There have also been related efforts in improving the efficiency of incremental compilation. Systems like Magpie [6], Pecan [49], etc. have demonstrated that this can be accomplished in a fairly efficient manner.

Systems, such as CPS, have demonstrated the potential for human-
engineering through tightly integrated tools. Tools in these systems have all been designed with a knowledge of the structure of the system-wide information. They have also been designed to manipulate information in this base in an incremental manner. Most existing tools embody neither of these architectural principles. Consequently any level of integration that is possible cannot be as "tight". In this paper we will detail an alternative approach, called loose integration, which can provide the users with the type of focused support indicated above, with existing tools. In many instances these existing tools in a loose integration can be enhanced to provide some services that the more technologically advanced tools, e.g. CPS, cannot supply.

The object of this research is to improve the power and usability of existing software tools by integrating them into a software environment. A detailed examination of other systems, including those mentioned above, and the support which each of them provides for tools is necessary for contrast and comparison. Ideas of both a positive and a negative nature can be gleaned from such scrutiny.

3.1. Operating Systems and Concepts

The Unix\textsuperscript{3} operating system \cite{15}, although not a complete programming environment, has contributed some important concepts in the use of software tools. Pipes are used to permit two tools to communicate by causing the output of one tool to become the input of the other. The key to the importance of this concept is that the pipe can be established dynamically by the user via a simple command to the operating system.

\textsuperscript{3}Unix is a registered trademark of Bell Labs.
This pipe concept is useful only because all the standard tools in the set of Unix tools conform to the input/output protocol of a stream of character. This allows the pipe mechanism to connect these tools without concern for compatibility.

The fact that the input to, and the output from, a tool can be redirected to another tool adds much flexibility to this Unix environment. The user has more control over the operation of the tools. (Some simple uses of this redirection can illustrate its power. If a tool is going to be used repeatedly with the same input, perhaps during testing, that input can be inserted into a file and the tool's input redirected to that file. Conversely, if a tool is producing a large volume of results which need further study, the tool's output can be redirected to a file.)

The Unix tools are purposefully designed to be limited in their application rather than general purpose. That is, each tool performs a single, simple task. Larger tasks are accomplished by various combinations of tools connected by means of pipes or through input/output redirection. This can be characterized as a tool-kit architecture.

The interactive command language (shell) furnishes an easy mechanism for providing sophisticated control of the tools. Algorithmic control of the tools is possible using the programming language constructs of this shell. Operating system commands can be embedded in C language program in order to provided complex control for tools. This includes the use of the pipe and input/output redirection concepts.

The Unix operating system provides the user with a range of commands and functions to control the operation of subprocesses. These
facilities are useful for the integration of existing tools into the Unix environment.

There are two primary limitations of the Unix operating system with respect to tool integration. These are the lack of structure in the pipe mechanism and the lack of incrementality of the tools. The pipe conveys a stream of characters between the tools. Any structure in the information must be coded in terms of this set of characters. Both tools must be aware of this encoding. Furthermore, the sending tool must take the time and effort necessary to code its output, while the receiving tool is forced to immediately reparse this if that structure is to be used. As an example, consider any tool which produces tabular output of several fields. The sort tool can then be used to order the rows of the table with respect to a particular field. When these two tools are connected via a pipe, the first tool converts its table to a stream of characters by including some additional data to indicate the structure of the table. The sort tool must then reconstruct the tabular form of the data before sorting on the required field. Thus the character stream protocol mandates additional processing which would be unnecessary if the pipe could convey more structured data.

The tool-kit architecture consisting of small, simple tools provides one step in the ability to integrate tools. Another step could be taken if the Unix tools were incremental in nature. This would increase the responsiveness of the tools to the user since incremental tools are able to use work accomplished on previous invocations of the tool rather than reprocessing the entire set of data. The standard Unix tools have not been designed to function in this manner.

The Smalltalk system [16] is one of the most successful of the object-
based operating systems. The object-based paradigm for programming affords an isolation of private information since each object can encapsulate its own data. The advantages of this information hiding are well known. In addition, the use of objects effectively decouples the entities of the system from one another. This simplifies the system by reducing the number and the complexity of interactions.

The class-subclass mechanism of Smalltalk and other such systems support incremental development of software. (This should not be confused with software that operates in an incremental manner.) A subclass can inherit any necessary data or operations from the class hierarchically above it. Thus, re-use of existing components is facilitated since systems can be developed in an evolutionary manner. Simple objects are further developed to complex ones. The inheritance of previously tested operations simplifies the testing by restricting it to the newly developed operations.

Tools in this type of system are objects. This correspondence forms a nice conceptual foundation for the user. The object nature of the tools is congruent to the user's image of those tools. The concept is often extended by representing the tools visually as icons. Tools are then controlled by manipulation of these icons.

The human engineering advantages are further extended by the use of windows to represent tasks and by the non-preemptive method of moving between tasks by positioning the cursor in another window. Thus the user can switch tasks by changing his/her focus of attention. He/she can later return to the original task by returning the cursor, which is a reflection of his/her attention, to the original window.
Object-based systems seem to have their prime influence and usefulness in exploratory programming or prototyping. The development of new software which is built from previously constructed programs is quite simple. The prototype can quickly arise from the remains of earlier attempts or previously developed systems.

Conversely, traditional program development is not well supported. An advertised advantage of an object-based system is its weak type checking. The type of messages sent to, or results produced by, these objects can vary dynamically. The lack of strong type checking, while advantageous in exploratory programming, is a disadvantage in the use of more traditional methods of project development. It is difficult to support the programming methods which have become standards for efficiently moving a project from requirements specification, through design, to coding and maintenance. Furthermore, existing compiler technology cannot be applied to provide compile-time error checks. Because of the inherent flexibility, most errors produced are run-time errors. These, of course, tend to be the most difficult to correct.

The integration of existing tools into an object-based system is quite difficult. All of the tools of such a system are required to function as objects of a particular format which respond to messages in a special style. This may require extensive changes in an existing tool to adapt it to this object paradigm. (Conversely, if a tool already meets those object-based requirements, then its integration into this type of system is almost trivial.)

The above audit of current operating system concepts has produced some which have proved useful for the integration of tools. These have included the pipe and I/O redirection constructs of Unix and the
information hiding and local data store concepts of the object-based systems. On the negative side, the lack of structure in the pipe mechanism was recognized as a major hindrance to its use in tool communication. The lack of incremency of the tool is a fait accompli of the existing tools of Unix (although Unix seems to provide no inherent restriction to incremental tools.) The major limitation of Smalltalk and similar systems is the lack of support for the traditional, well developed methods of project development which, in the current technology, seem to be mandatory for the successful completion of large, complex software projects. Until the exploratory method of programming is further developed or improved, the utility of object-based systems for large project development will be limited.

3.2. Programming Environments

An important advancement in the area of software engineering (with ramifications in the realms of programming languages and operating systems) was the development of programming environments. This term refers to software systems which aid the programmer in the task of program development by providing a friendly interface, giving rapid and more complete response, and furnishing needed information in a timely manner.

Prior to discussing some of the important, contemporary programming environments, it is useful to focus on and define some of the concepts which tend to recur in many of the environments, particularly the more advanced systems. The first such concept is the use of a common database for the collection of all project information. This repository is organized in specific formats which vary among the environments. In each case, though, it provides a unifying medium for the tools to store
results and from which to obtain input. This common base permits the tools to function in a more integrated manner. The integration of tools, the second important concept, results in tools which communicate and cooperate toward a common goal. The cumulative effect of these tools in cooperation is generally greater than that of the tools individually. In addition, a level of user friendliness can be produced by such an integration. The third concept is the idea of incremental tool cooperation. A tool that functions incrementally can give the user quicker feedback and more control of the process. This occurs because the tool is able to re-use some of the work accomplished previously to ease its current task. For example, by using a symbol table and parse tree produced on a previous invocation, a compiler can more effectively and efficiently translate the changes introduced into the program. Various combinations of these three concepts form the primary advancements represented by the following state-of-the-art programming environments.

3.2.1. Programming Environments Based on a Structure Editor

One significant group of programming environments developed over the past few years have centered on the use of a structure editor. A structure editor is a program which facilitates changes in a document (usually a program) by manipulating, under the direction of the user, syntactic units of that document (program). The editor is cognizant of the language in which the program being manipulated is written. The result is programs which are free of syntax errors or in which such errors are identified while the user is creating the program. The human engineering advantages of this type of system are significant.

The programming environments whose discussion follows are all based on structure editors. The implementation and usage varies across the
range of these editors. Besides sharing the advantages of an editor which is aware of programming language structure, these environments have some mutual limitations. As programming environments, they effectively support programming-in-the-small, but accomplish little in the area of programming-in-the-large [8, 9]. To pursue this, no sustentation is given to a programming method that could guide the programmer from requirement specification, to design, and coding. The project information base consists of programming information alone. No attempt is made to collect data from other phases of project development.

These environments are generally limited to the structure of one particular language. Some have been extended to other languages by means of an editor generator. Those which have not been so extended are quite restrictive.

This set of environments are generally not extensible by the user either in the sense of integrating additional tools, or in adding new commands. The environments' tools are so tightly cooperative that inserting additional tools or commands is a difficult task. The level of integration is so tight as to, at times, give the user too much information. The level of feedback is inflexible. (Some structured editors, e.g. the Cornell Program Synthesizer, have attempted to solve this by the use of ellipsis. Certain levels of the program are made invisible except for the comments which help explain these missing portions.)

The Cornell Program Synthesizer (CPS) [66, 65, 51] was the vanguard in the development of the structure editor concept. By maintaining a parse tree as the programmer is editing, the task of the interpreter is made much less complex. The parse tree functions as the common project information repository which is shared by the primary tools - the
structure editor and the interpreter. These tools have an embedded knowledge of a subset of the PL/1 programming language.

The user interface to CPS is template-driven. Programming proceeds by a sequence of user commands that produce programming language templates and that permit the insertion of variables, expressions, etc. necessary to complete the template. The user is not permitted to change the individual keywords of the template, but must manipulate the construct as a whole. Only syntactically correct programs are permitted (except for a few errors that are more conveniently noted by highlighting). This feature is enforced by parsing and interpreting each component as soon as manipulation of it is completed. Incomplete programs can be executed since the parsetree is maintained and an interpreter is available. When a run-time error occurs, control is returned to the editor at the point of the error.

The Magpie system [56] focuses on the concept of incremental compilation. During the think time when the programmer is not editing, the Magpie system uses the computer cycles that would otherwise be wasted to compile the code that has been introduced or changed since the previous compilation. An incremental analyzer maintains an attributed parse tree which is incrementally converted to object code. The linking and loading are similarly performed incrementally.

This programming environment is treated as a single tool rather than a collection of tools. The system's authors assert that all commands which are appropriate in a given context are legitimate. Rather than a single database, Magpie works with two entities, the source code and the execution state. The user is permitted to view both of these through browsers. The syntax and the static semantics of a procedure of the
source code are incrementally analyzed. When this step is completed, the procedure is queued for translation which occurs when there is a break in the user's input. Execution and debugging functions are implemented by instrumenting the actual code with debugging code.

Magpie is text-oriented as opposed to being template-driven. As such, it is not a true syntax directed editor, but uses the incremental compiler to accomplish a similar result. The unit of incrementality for static analysis is a single character. As static errors are discovered they are highlighted. Since only the first such error can be discovered by this text technique, Magpie breaks the program into logical fragments which are separately compiled. Thus, Magpie provides language directed editing in which the syntax and static semantics are incrementally analyzed. But no attempt is made to prevent syntax errors or to automatically correct them.

Statements may be executed in the context of a specific activation of the execution stack. Magpie provides a workspace in which to enter these statements to be executed immediately by the "do it" command. These function as a convenient mechanism of testing portions of a program. The statements inserted into the workspace can act as a driving routine to call other procedure in the current execution state.

Event monitors, called "demons", can be prescribed for the system. These demons are then invoked whenever the specified action occurs. These event monitors are useful for providing automatic processing to support the user. Possible uses include restricting access to portions of the project information, providing some useful information to the user upon the occurrence certain events, or sending reports to a manager on a user's progress.
The Magpie system is designed to operate in a workstation environment. It functions well by using the resources of the workstation which would otherwise be wasted to incrementally compile the partially completed program. This approach may not work as effectively in a time sharing environment in which excess processor cycles are used by other users.

The Pecan programming environment [49] uses an abstract syntax tree as its common, unifying data structure. Various views of this are separately maintained for the user. The source code is one view of this tree; the execution environment is another separate view. These views are coordinated and kept current by means of message passing. (The maintenance of these separate views and the use of message passing to synchronize them is an implementation decision that may increase the response time of the system, and consequently reduce the number of tools which can effectively be integrated into the system.)

The language-based editor is accessible to the user through three separate modes. The simplest to use for the novice is a set of templates presented by means of a menu. Typing of the full text of the programming construct is also permitted. The expert is likely to use and appreciate the capability of typing the prefix of each template entry.

The Mentor programming environment [10] is also founded on an abstract syntax tree as the common information base. This syntax tree is annotated by means of other abstract syntax trees in specialized languages. The main philosophy is to build specialized interpreters which aid the programmer by performing various computations and rearrangements of the program. These interpreters communicate through the abstract syntax of Pascal and its annotations. For example,
debugging is accomplished by means of special versions of the source program with built-in user interfaces which are compiled on the standard compiler.

Mentor is designed to be open-ended by allowing arbitrary annotations of nodes of the syntax tree by abstract syntax trees in a specialized language, i.e. by maintaining various views. The user can execute in coroutine with the program to allow controlled manipulation of the program.

The Pascal Oriented Environment (POE) [13] is a programming environment based on a structure editor which is cognizant of the syntax of the Pascal programming language. The program is maintained as a text file rather than as a tree. This has the advantage of permitting other tools to use the programs produced and of allowing POE to read programs in text files that it did not create. Conversely, this coarse level for the common data base means that extensive translations are necessary to provide the syntax and static semantic checking desired. POE uses a standard Pascal compiler to translate the completed program. (This is an attempt to permit the integration of existing tools by using a text file as the communicating medium. In the research described by this dissertation, the approach has been to allow a more sophisticated database to be shared among the existing tools. View extractors and output distributors convert from this common data store to the syntax required by the tools.)

The POE user interface is based upon typing token prefixes. An automatic error repair algorithm is used to fix errors as they are detected and to produce a general "undo" mechanism. Full static semantic error checking is provided with the highlighting of errors when
they are discovered. Positioning the cursor on the error produces a
descriptive error message. The implementation strategy involves the use
of an extension to attribute grammars which allow attributes to flow
directly to the location of use, rather than to follow the parse tree paths.

Arcturus [62, 61] is a tight integration of an editor, an interpreter, and
a debugger which is based on the Ada programming language. The user
interface is template assisted for Ada text editing. An Ada Program
Design Language is automated to aid in the stepwise refinement from
program designs in this PDL to Ada programs.

The DICE programming environment [14] has a different flavor as it is
designed to aid in program development in a distributed environment. In
particular, DICE is designed for developing a system on a host machine,
but which is to be executed on another target machine that is connected
to the host. The host is to be used for remote debugging and
maintenance. An incremental compiler maintains a database of cross
references and static analyses. Programs are represented as abstract
syntax trees. The compiler supports statement-level recompilation by
means of these extra databases. This database also provides the
information necessary for debugging.

3.2.2. Programming Environments which Supports the Life Cycle of
Program Development

One of the acute shortfalls of the programming environments discussed
above is the lack of support for the entire life cycle of program
development - from requirement specification, through design, to coding.
The following environments attempt a wider spectrum of support. Each
has its own forte for which it supplies more substantive aid. But all tend
to give some assistance to project development at several levels,
These systems, while providing support for a greater percentage of the program development cycle, tend to give less specific aid at each level. Another shortcoming that this group of environments shares is the lack of support for a programming method. The process of designing and coding complicated programs is such a complex task that it requires the uniform application of a standard set of guidelines and rules by all members of a project development team. To insure the uniform application of this method, it should be enforced with the aid of the environment.

Conceptually, the prime shortfall of this group of environments is a general lack of a model to underlie all the project information. The structure editors discussed previously were typified by the use of an information base structured by the presence of an supporting model. This often takes the form of a parse tree of an abstract syntax tree. In either case the supporting model is usually a grammar for the programming language being used. The group of systems currently being described have a more generalized focus in attempting to support several phases of project development. With this goal, it has not been clear what all encompassing model could be used. The lack of such a model reveals a critical difference in the kind of specific processing that the previously discussed programming environments are capable of performing for the user as opposed to the more general tasks performed by the environments of this section.

Argus [64] uses a centralized project information repository consisting of a relational database and a collection of files. The customizable editor uses coding templates. The user's status is saved when he/she leaves a component of the system. This allows the user to subsequently return to
that component with its status restored. New tools can be integrated by changing the user interface. Because of this method of integration, a loose level of integration is all that is possible. Furthermore the shared database consists of a coarse granularity which restricts the degree to which sharing and cooperation can occur among tools.

The Interlisp environment [67] is a successful system based on the Lisp programming language. As a residential system, a primary copy of the program and of the tools must reside in memory. Masterscope is a component which analyzes and cross references user programs to predict the results of a change or to make a pervasive change throughout the program. The DWIM (Do What I Mean) component makes use of contextual information to interpret the user's requests. The tools of Interlisp are integrated by the fact that, as a residential system, the tools can call each other without termination (in the corrotine model) to preserve the context. The Programmer's Assistant is a component whose set of actions includes history, undo, redo, and use commands.

The residential nature of the system, while allowing for easy communication among the tools, may be restrictive even on a virtual memory machine. The sophistication of modern tools results in system components which are voracious in their appetite for system resources, particularly for memory.

Toolpack [40, 41, 39] is an environment which provides portable, extensible support for the development of Fortran programs. Each tool has an embedded knowledge of the syntax of Fortran. The collection of tools is integrated by the tree structured file system which serves as the centralized data repository. The tool set is actually a collection of tool fragments which can be composed in various ways by means of a
command language. Replacing tools with upgraded versions is easy provided that the new versions take the same input and produce the same output files. Toolpack maintains a dependency DAG for the tools.

Toolpack's prime limitation are in the coarse granularity of the information base (i.e. the file system) and the lack of incremental tool interaction. The use of a file system as the information repository makes the initial process of integration of additional tools less complex. But the result ia a gross level of communication in which each tool operates in a monolithic manner in completing its use of the input file before releasing its results in the output file. The response to the user is less rapid as he/she waits for the completion of a tool's operation on its files. This problem is exasperated by the lack of incrementality in the operation of the tools.

The Program Development System (PDS) [4] is particularly useful for environments which support the development of a family of programs. This capability to support a group of related programs is provided by its maintenance of revisions and versions of programs. PDS has an interactive editor which directs commands to components and records effects on the program stored in the database. The common data structure used by the system is a computation tree represented as a list structure. PDS attempts to support the design and implementation of software by storing the abstract design of the program, the concrete implementation, and the transformations that refine the abstract into the concrete. Thus, if the abstract representation is changed, much of the concrete can be restored by application of the transformations.

The domain of PDS is limited to the design and coding phases, while ignoring the requirements specification and high level management
decisions. PDS was designed as a self-contained unit that does not lend itself to the integration of other tools.

The Gandalf environment [17] is a comprehensive system providing support for the entire life cycle of software development. System version control components attack the problems of programming-in-the-large while incremental program construction tools aid programming-in-the-small [8, 9]. In addition, project management components provide support for programming-in-the-many (a term coined by the Gandalf group to describe the tasks of supporting multiple project members).

The Gandalf system is integrated in that each component of the system is knowledgeable about the other pieces. The common development database is used to capture project information of such diverse types as source code (in the C language), access lists, and version control information. All objects in the environment are manipulated through a uniform interface developed by ALOE (A Language Oriented Editor) [35]. Action routines are implicitly invoked when the user applies particular manipulation commands to specific types of subtrees of the project information. These are used to automatically check the semantics of the program being developed, check the consistency of the module interfaces, and to support windows for various contexts, etc.

Gandalf's primary limitation is that no support is provided for a programming method. As discussed previously, such a method is a mandate for successful development of a complex project. Another limitation is that no aid is given for the integration of existing tools.
3.2.3. Artificial Intelligence Concepts and Systems

The development of programming environments will be influenced in a continually growing sense by concepts and techniques from the realm of artificial intelligence. This is particularly true as more experience is gained in the application of software development methods. To develop an "expert system", it is requisite to have a human expert from which to glean the necessary knowledge. To this point, although there are a great number of software engineers, there seem to be little consensus as to the optimal method of software development which would fit the majority of development environments. Some of the important concepts which have proved useful to this research are the following:

- the use of a knowledge base [38],
- the separation of knowledge and control [24], and
- the use of procedural components with local "experts" knowledge.

The first two of these are related and can form a major support for the implementation of a successful programming environment. The knowledge about the manner in which the project information is stored is critical information. In the TRIAD system this knowledge is captured by the attributed grammar form [34] which structures the project information. This knowledge facilitates many methods of analysis of the project information that would otherwise be virtually impossible. The separation of knowledge from control allows either entity to be changed or modified with a minimal effect on the other.

The use of local experts, sometimes called demons, is an
implementation technique which simplifies many kinds of processing. By associating procedural components with localized portions of the project information, the processing necessary is made less complex. The local expert has a detailed knowledge of and access to the context in which it resides. Because its task is restricted to a small, localized context, its complexity is generally reduced. Furthermore, changes to the structure of a portion of the project information repository affects only a few of these local experts. The ramifications of modification are isolated and restricted.

One of the primary conceptual underpinnings of the TRIAD software environment, that of information frames is a confluence of ideas from the artificial intelligence and the software engineering domains. Software engineers have long insisted that information is the most important resource that an organization possesses. This concept has been executed in TRIAD by the storage and structuring of all project information in the information repository. The artificial intelligence frames approach has influenced the method of structuring or organizing this information. Frames provide a means of attaching attributes to particular pieces of information and of sharing these attributes to related chunks of information in an inherited manner. Therefore, frames furnish a method of organizing information stored in a template-like mechanism (slots) about an object or class of objects. This organizational or structural data in the form of links (e.g. the "IS-A" link) enables the inference of information. In a similar manner, the TRIAD system provides structural links between logical chunks of project information which is stored in a template-like form. This enables the recovery of inferential information in an analogous manner.
The TRIAD use of information frames is an extension of the artificial intelligence frame. In a traditional frame, the instances appear at the leaf level of a tree of frames. The non-leaf nodes provide attributes that describe the concept represented by each frame. The leaf nodes each inherit all the attributes of their ancestor nodes (although exceptions for a particular instance can be noted). The TRIAD information frame allows for instantiation at all levels. The slots no longer represent information that holds for all instances of a particular type of object, but are used as tags to organize the project information. (Slots in the traditional frame are unlabeled and are used as attribute values.) The links between information frames generally represent a refinement concept - the child node is a more specific representation of a portion of the information of the parent node.

The traditional frames are concept oriented. One tree of frames represents one concept. The information frames are more process oriented. A tree of information frames describes the process of project development.

Information frames form a conceptual framework for the organization of project information which can easily be used for information storage, retrieval, and analysis. This concept has been implemented by the use of attribute grammar forms [34].

The discussion to follow highlights the major features of some of the existing research systems which can be considered to reside in the artificial intelligence domain. The first is the Programmer's Apprentice [52]. This system was designed as an aid to the programmer by allowing the editing of a program as an algorithmic structure, rather than as a syntactic one. The Apprentice records details and assists in
documentation, verification, and modification of a program, while the programmer accomplishes the more difficult parts of design and implementation.

The major components of the Programmer's Assistant are Plans, Analyzer, Coder, Drawer, Plan Library, and Plan Editor. Plans are the descriptions of the algorithmic structure of the program. These are created via the Plan Editor and are stored in the Plan Library. The Analyzer constructs a Plan from a program, while the Coder does the inverse. The Drawer can produce a visual representation of a Plan.

The Programmer's Assistant is limited in that it is not, by design, a complete programming environment, but rather augments the environment. In addition, support for a method of project development is lacking. The important problems of project management are not addressed.

The idea of using frames to represent knowledge, as discussed previously, has the value of storing local information in its context. The FRL system [54] provides a mechanism for the use of frames. The use of FRL for frames is, of course, an implementation technique, not an environment for the support of programming. But the technique is a useful one, despite the fact that there are some limitations to its use.

Omega [30, 43] is another knowledge representation system. Its primitive concepts include inheritance, instantiation, viewpoint, and logical operations. It is a self-describing system since it can represent its own rules of inference.

A example of a knowledge-based tool for supporting the development
and maintenance of software is the Intelligent Program Editor (IPE) [12]. This system uses an external model of the programming process. A database, the Extended Program Model, represents the functional structure of the code. For example, the knowledge base includes information about data and control flow by means of a parse abstraction. The knowledge base provides ways of associating intentions behind a program with specific features of the code. IPE is again limited in providing no aid for a method of project development. In addition, the system is not easily extended or adapted.

The above consideration of existing systems spanning the areas of operating systems, programming environments, and artificial intelligence has illustrated useful concepts and techniques that may be productive in creating powerful and user friendly programming environments. The shortcomings of these systems are also a source of experience which points to areas that need further attention in subsequent environments.

3.3. Other Work on Tool Integration

The past decade has witnessed some work of both a practical [2, 3] and a theoretical [31, 32] nature in the integration of tools across a distributed, heterogeneous system. This work has centered on achieving a uniform access interface to various tools located at diverse sites in the distributed system. The user references objects (files, tools, etc.) by means of a global name. The integrating system provides the user with access to this entity while hiding the details of the physical location of this object in the distributed system. The problems to overcome include those of global addressing and of information representation and conversion. There is no attempt to increase the power of the tools or to allow the tools to communicate in more sophisticated manners. Since
there is no model (or method) underlying the canonical information representation used, many of the techniques given in the research described by this dissertation can not be applied to this distributed environment.
Chapter 4
The Parameters for, and a Categorization of, the Degree of Tool Integration

4.1. Components of the System

Any software development environment and, in particular, an environment that supports tool integration consists of the following logical components:

- User interface or monitor,
- Tools, and
- Project information base.

An analysis of existing systems reveals the presence of these components although they may not appear as distinct units but are distributed across the system.

4.1.1. User Interface Design

Each of the terms "monitor" and "user interface" describes an aspect of the multiple roles of this component. There is indeed the function of providing an adequate and consistent interface to the user community. The user's view of a software environment is colored most by the interface that is used to access the project information base and the tools. To the user, this interface is the software environment. By
prompting the user and by monitoring the user's progress, the interface should effectively guide the user in the uniform application of a method of project development that follows company and project standards. The result is a project-wide (or even company-wide) standard application of a software development method.

The term "monitor" is more apropos in describing the task of overseeing the user's interactions and auditing the operation of the tools and the information base. In an unintegrated environment each tool provides its own interface with its own syntax and modes of operation. A user interface should also be a conduit, or channel, of information from the user to the project information base, and vice versa. It must monitor and structure this flow of information to enforce company standards and methods. As a controlling agent, the monitor interprets requests and commands from the user and passes them on in the syntactically correct form to the appropriate tool. This helps to smooth the transitions between tools and to alleviate the modes of operation that are inherent in many tools. (Tools with modes of operation are those in which the validity of commands depends on the state of the tool.) The user interface should coordinate the use of the various tools. The monitor is in the position to smooth the transition from tool to tool, and to even disguise the tools so that the user is not distracted from the portion (or chunk) of project information on which the he/she is concentrating. The monitor also produces valuable status information about the users' interaction with the system and with the tools. This is useful to a system manager in analysis of the operation and the efficacy of the environment. As the monitor is endued with more intelligence to become an expert system, it can use this status information to automatically tune the performance of the system.
4.1.2. Software Tools

The necessity of adequate tools in a software environment has long been recognized. There is a real sense in which all of computer science is founded on the premise that tools should be used to manipulate symbolic and numeric data. The particular tools to be included in a software environment depend somewhat on the type of software being developed. A small assembler shop has little need for the version control or optimizing compiler needed by a software house. A software house has very little need for a hardware simulator need by a defense contractor. Furthermore, tools evolve and new tools are developed which eclipse the former ones in both power and user friendliness.

4.1.3. Information Base

The project information base can be considered the fulcrum around which the remainder of the system pivots. This information base provides an organized store for all project information. Not only should the code produced by the project members be recorded in the information base, but also requirements and design decisions, management information, version control data, etc. must be maintained. The point has been made and frequently observed that design and implementation decisions and the rationale for those decisions are often lost through failure to record that data in a well organized repository [46, 25, 26]. It is vitally important that this information be easily retrieved for later use in making changes, in maintenance, or in developing similar projects.

These constituents form the focus of the ensuing discussion. The parameters of tool integration pertain to these components.
4.2. Parameters of Tool Integration

The complex interactions of sophisticated tools that often arise when the tools, the information base, and the monitor are integrated can be characterized by three parameters:

- granularity,
- cohesion, and
- harmony.

The granularity of an object or of a concept refers to the size of the components of that entity. It ranges in a continuum from coarse, meaning large chunks, to fine, or small, chunks.

Cohesion is synonymous with structure. The greater the cohesion of an object or concept, the more substantive is the structure relating the components of the entity. An apt analogy can be made with the purpose of "glue" in the construction of a child's plastic model car. This glue provides the correct relationships among the parts. Without it the parts fall into an amorphous mass. Cohesion describes the degree to which the structural glue is present to mold the components into a synergistic whole. In a programming environment this cohesion generally takes the form of information relating the various elements of the system.

Harmony reflects agreement between two elements of the programming environment. (The lack of agreement is termed dissonance.) Thus, two system components which have the same granularity of interaction are harmonious with respect to that granularity. Conversely, if the information base and a tool operate with different levels of cohesion of
the information that they use, they are dissonant regarding that cohesion.
A high level of tool integration is easier to attain among components
which are harmonious. The thesis of this research can be restated in this
terminology as follows:

It is possible to integrate existing tools into a software environment in
which the components display some degree of dissonance, i.e. even when
harmony is not present.

4.3. Categorization of Tools for Integration

The precise meanings of the previously described parameters or
attributes are determined by their application. These three parameters
will be brought to bear on each of the three system components -
monitor, information base, and tools. Some of these cases will be more
apropos to the description of tool integration than others. All will be
mentioned with special emphasis on those that pertain to categorizing
tool integration.

4.3.1. Granularity

The granularity of the information base describes the size of the
logical pieces of project information which compose the totality of the
information. The granularity is fine if the user and/or tools can access
small chunks of information. A strong analogy can be made by
comparing the granularity of a database system and a file server. The
database generally has a finer granularity since the groupings of
information to which it has access are records which are relatively small
chunks of information that may be spread across several files. Conversely,
the file server has access to larger grouping of information in the form of
files. Hence the file server has a more coarse granularity of information. A coarse granularity for the information base does not imply that the user or a tool cannot find the logical piece of information needed. The search for it will just be broader and will have less automated help.

Tool granularity has several valid interpretations. The first is the granularity of the information used or produced by the tool. This complements the previous concept - granularity of the information base. Most traditional compilers, for instance, use a file, the source file, as input and produce another file as output. If a valid symbol table or parse tree were available (perhaps as the result of another tool), a traditional compiler could make no use of it since its granularity of input information is too coarse.

Another view of tool granularity is that of the granularity of tool interaction with the user. A batch oriented tool has the most coarse granularity possible, i.e. no user interaction. A screen editor is typical of a highly interactive tool. This view of granularity, while valid, adds little to the categorization of tools for integration. The appropriateness of the level of granularity of interaction depends on the purpose of the tool and has little direct effect on its ability to be integrated into a software environment. (The presence of interactions with the user can be used as a technique to achieve a tighter integration of the tool. This concept will be amplified in a later chapter on integration techniques.) The granularity of interaction will be discussed in the context of the user interface where it is more significant.

An important aspect of tool granularity is the granularity of incrementality, i.e. the degree to which the tool operates incrementally. An incremental tool is one that accomplishes its task in small steps or
increments. By recording its internal state, it is able to avoid redoing work begun on a previous invocation. This is related to granularity of information of the tool in that an incremental tool most often consumes and produces information with a fine granularity. This information granularity, although important, is not the feature that determines the degree of incrementality. The crucial factor is the granularity of control of the tool. The control algorithm for an incremental tool is organized to operate in small increments rather than requiring to be completely executed at each invocation. In the context of an interactive tool, this incrementality requires more than the ability to execute a single command per tool invocation. The control algorithm must be designed to partially execute that single command, then to store its internal state in order to finish the execution later. A fine granularity of control, i.e. a high degree of incrementality, is generally useful in achieving tool integration. Techniques for overcoming a lack of such granularity will be discussed in a later chapter.

Granularity of the interface has one chief interpretation - the granularity of interaction with the user. Upon initial examination this would seem to be a characteristic of each tool. But the primary factor is the granularity of interaction provided by the user interface. The monitor can adjust this granularity as it controls the operation of the tools by automatically providing some input to a tool. This input may be acquired from various sources - default values indicated by the user, information retrieved from the information base, or data produced by other tools. Thus, the amount of user interaction with the system is largely a function of the monitor. This is one situation in which fine granularity may not yield a more human-engineered environment. It is more optimal to achieve some middle ground in which the user has
adequate control over the development and manipulation of project information, but is not overcome by the necessity for an extensive quantity of input.

4.3.2. Cohesion

Cohesion of the information base pertains to the amount of structural information present to relate various pieces of the project information. A coarse granularity in the information base leaves few possibilities for this type of relational data. Conversely, fine grains of project information imposes the exigency of an increased amount of this structural glue. A strong analogy can be made with the normalization of the relation in a relational database. Normalization of relations tends to diminish the number of components or fields in an entity (finer granularity) while increasing the number of relations (higher cohesion). Tautologically, the greater the amount of an environment's cohesion in an information base with fine granularity, the easier is the integration of tools into that environment. The tools simply have a greater aggregation of more useful information to access.

Cohesion of a tool has a dual meaning. First is a concept parallel to that of information base cohesion, that is the cohesion of the information produced by the tool. A tool that produces small, logical chunks of information (fine granularity) can more readily be integrated into the environment. The most important feature of "pipe" mechanism of Unix [15] is that it has a fine granularity (the character level). The result is a set of tools that can easily work together. This level of integration can be increased if the information produced also embodies structural information to relate the grains of project information. This could then be classified as a "high-level pipe" through which information in two forms, project information and relational information, is exchanged.
A second application of cohesion with respect to a tool is the cohesion of control. As discussed earlier, a fine granularity of control is necessary for incremental tool operation. A cohesive control for an incremental tool shares much of the control information from one invocation of the tool to the next. This sharing of internal status is requisite if the tool is to avoid redoing work. Fine granularity of control means that the tool's control algorithm is organized to function in small steps. Cohesion of this control implies that information from one tool application is recorded in some manner (e.g. in the information base) until the next incremental application. To be truly incremental the tool's control algorithm must possess both qualities.

Cohesion of the user interface models the degree of information retention and sharing among the user commands. The monitor is the proper component to remember the context of a command. User commands can be simplified if default values or previously entered values are inserted where appropriate. Carried to its logical conclusion, this would result in a sophisticated DWIM-like [67] mechanism that not only searches a list of known commands to find the closest match for a user command, but also could examine the user's dynamic context. This includes such items as the last tool used, the parameters passed to this tool last, etc.

4.3.3. Harmony

Harmony or dissonance is a property of a pair of system components. Thus we will examine harmony/dissonance of three types: harmony or dissonance between the tools and the information base, between the monitor and the tools, and between the monitor and the information base. The first of these pairings bears upon tool integration to the
greatest extent. The remaining two will be described for completeness. The amount of agreement between tools and the information base also covers the interaction between tools since that tool to tool interaction occurs through the information base.

**Tool-information base interaction** has two components, a syntactic one and a semantic one. Syntactic harmony refers to the agreement in the information representation of these two components. In particular, this agreement should occur in both granularity and cohesion. Close harmony of this type leads to an easier integration of the tools into the software environment since fewer conversions are needed from one form to another. The information representations of two tools are said to be syntactically harmonious if each is harmonious with the information base. (This is a kind of transitive harmony.)

Syntactic dissonance must be overcome in order to integrate a tool. A mismatch in either granularity or cohesion of information results in the loss of critical information. Methods of surmounting this obstacle will be further discussed in the chapter on techniques of integration.

Semantic harmony/dissonance is more critical in that semantic dissonance must be overcome to integrate tools and yet general techniques for solving this problem are not easily specified. Semantic harmony describes the agreement in meaning between the information used by the tool and that used in the information base. This agreement-disagreement paradigm has at least three applications. In order of increasing importance (i.e. difficulty of surmounting), these are:

1. semantics of the information representation,
2. semantics of the level of abstraction, and

3. semantics of the project information itself.

The same project information can be represented in a descriptive manner or in a transformational manner. The semantics of the information is identical, but the meaning of the information representation is quite divergent. Management information is most often recorded descriptively in details delineating percent of project completed, resources budgeted, etc. Conversely, some version control systems represent versions by recording changes made to the previous version. This often results in a more efficient use of the storage capacity of the system than to record a description of all versions. In either method, the same meaning can be represented although conversion from one form to the other is more than a change in syntax. (This dichotomy has been discussed by others in the terms procedural and descriptive methods of information representation.)

The harmony or dissonance of abstraction of information produced or used by a tool and that stored in the information base affects the integration of that tool. This level of abstraction can vary from quite detailed to summary in nature. Project information may have resulted from analysis or synthesis. Each level has some utility for certain situations. If there is dissonance between a tool's level of abstraction and that of the information base, correction may be quite difficult. Converting from a low level of abstraction in which much detail is present to a higher level is possible. But conversion in the opposite direction is virtually impossible.

The most difficult type of dissonance to overcome is that of the actual
semantics of the project information base. If there is no common ground between two tools or between a tool and the information in the repository, significant integration cannot be achieved. (By significant, we mean cooperation between the tools.) Often the problem is not a complete lack of commonality, but rather that their junction is not apparent. An apt analogy can be drawn to the parable of blind men describing an elephant. These descriptions range from "shaped like a tree trunk" to "feels like a snake" depending upon which portion of the pachyderm's anatomy is nearest to each man. These description seem entirely incongruous only if one is unaware of their commonality. As our understanding of a particular field depends, concepts that previously seemed unrelated are often discovered to share a factor of commonality, a unanimity of purpose or a unity of causation. (Science, in general, is the search for these common causes.) Integration of tools whose commonality has not been identified is impossible to any significant extent.

Techniques for overcoming syntactic dissonance will be discussed in a later chapter. Overcoming semantic dissonance of any of the three types mentioned above is much more difficult and often impossible. Even considering the integration of tools whose information is semantically dissonant is premature until a deeper understanding of the commonality of purpose is achieved.

The user interface and a tool are harmonious if the view of the tool presented to the user is harmonious with the tool's actual operation. For instance, in Smalltalk [16], the user sees a tool as an object that receives and responds to messages. This object-based paradigm precisely models the operation of tools in this system. More specifically, the tool is represented visually to the user as a descriptive icon.
Harmony is achieved between the user interface and the project information base when the way in which information is presented to the user or is collected from the user is reflected in the structure of the information base. The information base of the TRIAD software environment [25] is structured as a tree of forms. This tree models the underlying development method which reflects the general order in which this information is presented to, and requested from, the user.

These last two areas of application of the harmony/dissonance parameter affect the user friendliness and efficiency of the system. Dissonance in these areas can be overcome by the view extracting technique described in a subsequent chapter. Harmony or dissonance in the first area (tool - information base) is that which most affects the integration of existing tools into the system and is, thus, of more interest in this research.

4.4. Categorization of Integration - Loose versus Tight

The confluence of these parameters (granularity, cohesion, and harmony) with their various applications at different levels to a collection of tools collaborate to produce a continuum of degrees of tool integration for a system. These terms can be applied to an entire system, in which case it refers to all the tools of the system, or to a single tool. The terms "tight" and "loose" integration actually refer to the extremes of the continuum. A system that uses the features of fine granularity and high cohesion of the information base and of the tools and syntactic and semantic harmony of the tools to achieve a high degree of inter-tool communication and cooperation is categorized as tightly integrated. The Cornell Program Synthesizer [66] is a quintessential example of a tightly integrated system. (A more detailed itemization of the ways in which this system satisfies these parameters will follow in the next section.)
The advantages that result from a tightly integrated system are numerous. Tools are more efficient since there is less duplication of effort. Since tools store their results in, and take their input from, the information base which has a fine granularity and high cohesion, the tool does not have to reparse that input to recover the structural information.

Secondly, the tool's response time to the user tends to be fast for incremental tools since the tool is taking a small step each time it is invoked. This enables the user to focus his/her attention on the project information and the task at hand rather than distracting the user by long waits for service. As pointed out in the Magpie system [56], the power of today's computer systems are sufficient that servicing users often leaves computer time (i.e. CPU cycles) available between user keystrokes and while the user is thinking. This time can effectively be used to incrementally apply tools. As a consequence, the user has more neoteric data.

Not only are tool results available more quickly, particularly in the case of incremental tools, but also those results are more accessible since they are stored in the information base. Other tools can be applied as necessary to analyze, summarize, and report these results. Furthermore, the status of each tool is more accessible to the user if it is also stored in the information repository.

These advantages are somewhat counterbalanced by a few problems with the tight integration of tools. The economics of such an integration may make it infeasible. A tight integration involves writing new tools or extensively rewriting existing ones. The marginal advantage gained in achieving a close cooperation of the tools may, in some situations, argue for a looser level of cooperation. The inflexibility of this degree of
integration (i.e. the user cannot adjust the increment size for incremental tools) and the volume of information available may create an environment in which the user, although surrounded by powerful tools, feels uncomfortable and manipulated and, hence, is less productive.

At the other end of the tool integration spectrum is loose integration. (The case of no integration does not appear in this continuum.) This is a degree of cooperation between tools in which the granularity is more coarse and/or the cohesion is lower, and in which the tools have some degree of dissonance. Although this level of integration initially seems less beneficial, many existing systems use it to great advantage. The Toolpack/IST programming environment [40, 41] provides a framework in which tools with an inbred knowledge of Fortran can communicate through an organized file system. The granularity of the information base (i.e. the file system) is quite coarse. The tools produce information in a coarse granularity. The amount of cohesion present in the tools and the information base is minimal. (An analysis of Toolpack will follow in the next chapter.) Yet Toolpack is able to provide a certain level of tool cooperation which eases the user's task.

The implementation described in this research would be categorized as a loose integration of a C compiler and a symbolic debugger into the TRIAD software environment.

For tools whose purpose requires little user interaction, e.g. many batch-oriented tools, loose integration is most appropriate. For any tools, a loose integration is often more economic. That is, tools that have a lesser degree of cooperation and sharing are less expensive to construct (because the number of interaction with other tools is limited) and require fewer computer resources to operate.
There are situations in which the tight integration of a system like the Cornell Program Synthesizer provides more automated control and feedback to the user than is desired. This is especially true when the user is in the prototyping mode. A loose integration may provide the ideal framework for nonrigorous, exploratory project development.

These have been examples at the extremes of the tight-loose continuum. There is a whole range of intermediate levels with various combinations of parameter values. No one level of integration can be said to be optimal for all classes of tools and software environments. Moreover, no one level is optimal for one given environment at all times or for all tools within that environment. Factors affecting the choice of degree of integration include economic considerations, the purpose of the tool, an even the user's mind set with respect to use of a tool. The best solution is a compromise in which the user has some influence or control over the degree of integration of a tool.

The evaluation of a specific tool to determine its capability to be integrated is a subjective exercise. Some general comparative statements can often be made and substantiated by a close examination of tools. But a precise, quantitative measure of the integrability of a tool is beyond the current state-of-the-art. Figures 7, 8 and 9 provide a set of questions (forming a simple method) to help determine this capability for a tool. The greater the number of questions answered in the affirmative, the greater potential for integration. (Note that these questions are always applied to a given context which includes a information repository of some manner and an user interface or monitor.)
Information Repository

Granularity

1.1 Is the information accessible in small logically related chunks?
1.2 Can a particular item of information be retrieved without searching through a large amount of project information?

Cohesion

2.1 Is there information which designates the relationship of one piece of project information to another?
2.2 Is there a model which underlies the project information?

Tools

Granularity

3.1 Is the tool interactive?
3.2 If so, does this interaction occur throughout the operation of the tool (or just to initiate the tool)?
3.3 Does the tool operate incrementally?

Cohesion

4.1 Does the information consumed and produced by the tool contain structural data to relate the information (e.g. a parse tree rather than a textual representation of a program)?
4.2 Does the tool share control information from one invocation to the next (e.g. data about which chunk of project information was last processed)?

Figure 7: An Informal Method for Evaluating a Tool for its Integration Potential - Granularity and Cohesion of the Information Repository and the Tools
Monitor

Granularity

5.1 Does the user interface permit frequent interactions with the user (as opposed to furnishing more monolithic commands)?

Cohesion

6.1 Does the monitor remember context form command to command?
6.2 Can the user easily re-use values entered on a previous invocation?

Figure 8: An Informal Method for Evaluating a Tool for its Integration Potential - Granularity and Cohesion of the Monitor

4.5. Analysis of Some Typical Existing Systems in Terms of this Categorization

Since the application of these parameters to determine in which category of integration that a tool belongs is such a subjective procedure, it is useful to examine some well known systems or system components with respect to this taxonomy. The method depicted in figures 7 8 and 9 is applied in figure 10. The questions of the method are answered in tabular form with references to the question appearing by number.

4.5.1. Unix Pipes

The pipe [15] is a mechanism provided by the Unix operating system to permit the coordination and communication of tools. A pipe transmits the standard output of one tool to the standard input of a second tool. The information passed between the tools is a stream of characters.
Tool - Information Base

Syntactic Harmony

7.1 Is the size of the chunks of information produced by the tool the same as that used in the information base?
7.2 Does the structural information produced by the tool reflect a portion of that present in the information repository?

Semantic Harmony

8.1 Is the meaning of the information produced by the tool contained in the information base?
8.2 Does the tool use data at the same level of abstraction as the information base?
8.3 Is the information produced and consumed by the tool represented in the same general manner as stored in the information base?

Monitor - Tool

9.1 Does the interface present the user with a view of the tool which is similar to the tool's actual operation?

Monitor - Information Base

10.1 Is the information presented to the user in a manner which reflects the structure of the information base?
10.2 Does the interface request information from the user in an order that models the information repository?

Figure 9: An Informal Method for Evaluating a Tool for its Integration Potential - Harmony versus Dissonance

Thus, the granularity of the information is quite fine. The degree of harmony of this granularity among all the Unix tools is the prime factor in the success of this pipe concept. Any of the standard tools in the Unix environment for which such a connection make sense can
communicate via this technique. The consequence is a flexible environment in which the user feels a level of control over the use of the tools.

The limiting aspect of the pipe mechanism is the low degree of cohesion of the information that is transmitted through the pipe. The two tools using the pipe must agree, prior to usage, on the method used to code the structural information in terms of characters. As a result, the first tool codes the information which is immediately parsed or decoded by the receiving tool.

The granularity of control of tools is not affected positively or negatively by the pipe. Pipes are unidirectional, but this presents no problem since a separate pipe can be established for each direction. If tools act incrementally, the pipe will not hinder this. Conversely, tools that are not incremental will not be aided in achieving such cooperation.

The level of granularity of user interaction with the tool is affected indirectly by the pipe. The receiving tool of the pipe obtains all of its standard input from the other tool through the pipe. Hence, that receiving tool is effectively blocked from interaction with the user. (Commands could be passed to it by the other (sending) tool provided that the sending tool was designed to do this.)

4.5.2. The Cornell Program Synthesizer

The Cornell Program Synthesizer (CPS) [66, 65, 51] is typical of structure editors. The granularity of information shared among the tools is in the form of tokens and, thus, is quite fine (although not as fine as a stream of characters). This fine granularity is supplemented by a high
level of cohesion provided by maintaining a parse tree. This eliminates
the need to reparse information each time that a tool is invoked. As a
consequence, CPS achieves a high degree of incremental tool operation.
The harmony of granularity and cohesion reached by the common use of
a parse tree makes this level of integration uniform across all tools
(which, in this case, are an editor and interpreter).

The major drawback of the integration level of CPS, as with any very
tight integration, is its inflexibility. The user has no control over the
amount of feedback he receives about errors or over the frequency of
translation.

4.5.3. Pecan

The Pecan system [49, 50] is one of several new and interesting
research efforts in the realm of incremental compilation. Pecan provides
much more than this capability since it is actually a complete
programming environment. The granularity of access and cohesion of the
project information are similar to that of the Cornell Program
Synthesizer since Pecan also maintains a tree of code. The differences are
that this tree is an abstract syntax tree, and that the code is
incrementally compiled rather than interpreted. Despite this repository of
information that could potentially provide a harmony of information to
be used by all tools, the developers of this system have made the
implementation decision to maintain separate views of that syntax tree
for the use of various tools. The incremental compiler is treated as one
view, the execution environment as another. These views are coordinated
by the passing of messages. The result is a system with fine granularity
and high cohesion, and the capacity for great harmony, but a certain
level of dissonance imposed by the implementation strategy.
<table>
<thead>
<tr>
<th>Question Number</th>
<th>Unix</th>
<th>CPS</th>
<th>Pecan</th>
<th>Toolpack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1.2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2.1</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2.2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>3.1</td>
<td>Yes*</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3.2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3.3</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4.1</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4.2</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>5.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>6.1</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>6.2</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>7.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7.2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>8.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8.2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8.3</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>9.1</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>10.1</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>10.2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

* The pipe construct can work with interactive tools only through the tool at the in-take end of the pipe.

Figure 10: Evaluation of the Integrability of Some Existing Systems and Components.
4.6. Application of these Concepts to Other Domains

4.6.1. Important Features of the Domain of Software Project Development

The domain of software project development from whose study these concepts and parameters have arisen seems to be particularly well suited to the successful integration of tools. One of the pervading characteristics of this domain which lends this success is the harmony of purpose of the tools. Each is focused on providing aid to the user (programmer, analyst, manager) in the creation of a well-defined object, the software product.

This harmony of purpose is enhanced by the common form used to represent the project information, i.e. textual. This domain is typified by a syntactic harmony. Even the alternative forms of information used by some tools (e.g. the parse tree, etc.) have a precise correspondence in a textual form. This text format permeates the entire range of project information - from high-level management reports and design documents to programming code.

The single, most important feature of this domain for achieving a useful integration of tools is a semantic harmony. When a domain is extremely well understood, processing in that domain can often be automated. For example, the conversion of an algorithm described in a high level programming language to machine language can proceed by machine with no human intervention. That conversion process is known precisely enough to be described algorithmically. In a few, very limited domains, researchers are attempting to provide automatic program construction from more abstract design description. This is currently possible only in a limited realm in which the type of program to be constructed follows certain patterns very closely.
The software development field in toto is far from such automation of program construction. This is true despite the accumulated wisdom and experience of over thirty years of project development. This field is too broad to attempt to find an all encompassing algorithm for constructing any program from a high level description. (This is not to imply that such an algorithm may not be discovered for wider sections of this domain.) research in software engineering has provided some very useful, but general, techniques or methods for constructing software. These methods (e.g. Jackson [], HIPO [], SADT []) are a set of guidelines or rules to help to structure and standardize the software development process. These are clearly not algorithms in that multiple interpretations are possible. various users apply the same guidelines with differing results.

These software development methods do reflect a great deal of knowledge about the process of converting imprecisely stated plans and designs into a tight algorithm to satisfy those plans. This knowledge, in the form of well developed methods, provides the semantic harmony which binds the various phases of software development and which makes the integration of software tools possible.

4.6.2. The Domain of VLSI Design

For the purpose of contrast and comparison, let us consider another domain in which tool integration is more difficult. The domain of VLSI design, testing, and implementation. is also characterized by a general harmony of purpose - that of developing VLSI circuits which satisfy the requirements. Recent years have seen the introduction of a plethora of very useful tools. These tools, although they share a commonality of purpose, lack the syntactic harmony of software development tools. There
are three general formats of information - procedural, textual and graphic. In each of these general formats, individual tools vary widely in their syntax requirements. These is no industry-wide standard for the representation of the graphic descriptions of objects which are critical data for many tools. Thus, communication and sharing between tools, which is necessary for integration, is difficult. A complicated conversion between each pair of tools must be constructed. A more complex problem is that, since there is no common form for all VLSI project information, an integrated database which avoids duplication of data is not feasible. (The various incongruent views for each tool could be separately maintained in one location, but this would result in duplication of data and consistency problems with no gain in tool communication.)

The root of the difficulty lies in the fact that there are many VLSI implementation technologies, each of which attempts to optimize a particular set of attributes (e.g. component size, speed, power, etc.). No one technology has emerged as the optimal one in all cases. In fact, the development of such an optimal technology is not likely in the foreseeable future. Any tool integration in this field must occur at the higher levels of design in which functionality, and not efficiency, is the primary concern. This is analogous to the use of various implementation languages in the field of software development. Each language has a realm of applicability in which it is most efficient. This analogy was most apt when efficiency of execution of software was the compelling force. With the subsequent reduction in hardware cost and increase in hardware efficiency, the choice of a language from which efficient code can be produce has become a minor consideration for the domain of software development. The optimization of the programmer's and analyst's time in design, coding, and maintenance has become of
paramount concern. A corresponding development has not transpired, and is most probably not forthcoming, in the domain of VLSI in that optimization of the physical attributes of the end product is vitally important. In fact, these domains, VLSI and software development, are complementary in the sense that, in order for programmers and analysts to be able to avoid issues of efficiency of operation of the software which they produce, the constituent circuits of the hardware upon which the software runs should operate as efficiently as possible. Therefore, VLSI designs, by nature, are forced to concentrate on optimization of the physical properties of the resulting circuit.

The analogy between the domains of VLSI design and software development also fails in that the life time of data in these fields differs significantly. The longest and often the most costly phase of software development is that of maintenance. Programs generally have a relatively long life expectancy although subject to various modifications and revisions. Conversely, VLSI designs are generally discarded after the circuit has been implemented. New circuits are usually not developed by modifying previous designs (except in the use of some standard components), but by creating new designs. The maintenance activity is not nearly as important in VLSI development as in software development.

For these reasons, the development methods which have enabled the tool integration techniques to provide enhanced operation of existing tools are more limited in the VLSI domain. This limitation is inherent in a domain in which development methods are focused on the technology required by the implementation. In software development, a single method can be developed to structure the process of software
development from requirement specification, and design to coding while incorporating management details. Because of the importance of the end-product technology at all excepts the highest levels of design, a similar method does not currently exist, nor is such a method likely. This limitation lies in the category of semantic dissonance. Methods of guiding portions of the design, testing, and implementation phases are currently being developed. But these have not yet evolved sufficiently to permit the identification of the semantic commonality necessary for tool integration. Furthermore, it is not clear that any tool integration is necessary or desirable, except at the more abstract levels of design.

4.6.3. The Domain of Business Data Programming

Tools in some subdomains of software development can be integrated even more completely and easily than in the domain as a whole. The field of business data programming is one such sub-area. This field is understood so well that very precise patterns of programming have been identified [57, 58, 44, 47]. These few patterns can encapsulate almost all the types of programs that are necessary in this field. This set of patterns forms an explicit method for the domain. Because of the precision with which these patterns can be specified, the tool support provided can be extensive as it develops towards automated programming. The semantic dissonance in this sub-domain has virtually disappeared. This harmony is not an inherent quality of this particular field, but is a function of the narrow domain of the field.
Chapter 5
Integration of Existing Tools

5.1. Environmental Requirements for Integration of Existing Tools

In order to answer the question "Can a given existing tool be integrated in a significant manner into a particular software environment", the environmental prerequisites need to be enumerated. This section addresses that topic.

5.1.1. Requisite Properties of the Operating System

The operating system which supports the environment must allow for a monitor which can retain control of the execution environment. That is, the monitor must have the capability of invoking tools, keeping account of each tool's status, storing data in, and retrieving it from, the information base, and interacting with the user. The operating system must permit this usurpation of power by the monitor.

The ability to integrate existing tools into the system under control of the monitor requires a multi-processing potential. The integrating tools can then be operated as a child process (or subprocess) with the monitor as the parent process. This property actually encompasses that of the previous paragraph if control passes back to the parent process when the child is terminated. The monitor, as the parent, can control the operation of the tools and administer the interaction between the user and the tool.
Another property implied by the above discussion is that the parent process (the monitor) should be able to trap the output of the child process (the tool) and to send appropriate input to the child.

5.1.2. Requisite Properties of the Information Repository

The necessity of an information base has been previously established. This underlying information base should provide a standard format for the project information. The mechanism for accessing that information should be flexible enough to allow the monitor to store the results of tools in the proper context. Most important is the ability to retrieve information on the demand of the monitor and, indirectly, of the user or a tool. The capability of retrieving small logical units of information (i.e. a fine granularity), although not a necessity, will facilitate the integration of tools by allowing the tools access to precisely the shared information needed without having to search through large chunks of project information.

This component has been described in a logical or functional sense as an information base. The choices for implementation vary from the use of a centralized file system as in Toolpack/IST [40, 41] to that of a relational database. The finer granularity and excellent querying facilities of most relational database systems make such a system an excellent choice. Other implementation schemes provide sufficient facilities although the task of tool integration becomes more difficult.
5.1.3. Requisite Properties of the Tools

To be integrated, a tool must not object to receiving its input from another source, i.e. from the monitor and indirectly from the information base. (The underlying operating system sometimes provides this facility for all tools.) Similarly, it must be possible to trap or redirect the results of a tool.

An interactive tool is one that can normally be more tightly integrated. The interaction with the user obviously gives the user a dynamic control that is vital to a human engineered system. But more than this, these points of interaction provide natural breaks in the operation of the tool that the monitor can use to support the user. While one tool is pausing for input, the monitor can invoke another, query the database, or provide other service that will aid the user.

An incremental tool provides a superb level of human engineering. The user is provided with incremental feedback at short intervals of time. There is no long delay for a more monolithic tool to perform its task. An incremental tool also provides a means for the monitor to supervise, control, and inform the tools. Thus, an incremental tool is, in general, a worthy goal in order for a system to be human engineered in an improved manner. Most existing tools have not been designed incrementally. Incremental compilers are just now becoming important items of research and development [49, 6]. An interactive tool can be made to simulate incremental behaviour by using the pauses for input effectively as described above.
5.2. Useful Concepts/Constructs in Achieving this Integration

To reach a high level of integration of existing tools, some concepts or constructs have proven important. These are:

- a monitor
- a method
- a view extractor and output distributor
- a database and attributes

5.2.1. The Monitor

The general purpose of the monitor (alias user interface) is to coordinate the various interactions, i.e. tool-tool, tool-information base, user-tool, and user-information base. The monitor can provide an important interface to the tools for the user. Since all interactions between the user and tools occur through the monitor, it can provide a more uniform interface by providing a common set of prompts and a consistent command syntax for all tools. The monitor can trap, process, and redirect the input and output to/from the tools.

As the controller of the tools, the monitor can manipulate the tools to produce a synergistic effect which is greater than that of any single tool. For example, the monitor can coordinate an editor and debugger so that when the debugger stops at a breakpoint, the editor's cursor is positioned in the project information at the corresponding statement.

The monitor can approximate incremental tool operation for an
interactive, but non-incremental, tool. When the tool pauses for user input, the monitor can usurp control. Then, before letting the first tool resume by providing input to it, the monitor can initiate another tool, obtain data from the information base, analyze project information produced thus far, etc. This process simulates incremental tool operation by providing information from multiple sources within a small interval of time.

The monitor can incorporate the information base to a greater extent in tool processing. Relevant project information can be retrieved to present to the user or to modify or enhance tool commands. For example, the project information at the code level can be queried to determine the locations of all statements that modify the variable "x". This can then be used to send several commands to the debugger which result in the setting of breakpoints at each of these locations. This task is beyond the scope of either tool working individually without user intervention.

5.2.2. A Method

The primary purpose of a method is to overcome semantic dissonance that occurs in a problem solving situation in a particular domain. A method is a plan for problem solving in a specific area of application. If a task in a problem domain is completely understood, its solution can be automated. (For example, computer programs can be compiled by machine because the translation process is well understood.) If a large body of experience exists in a specific sphere, a method can be developed to describe the general approaches depicted in this expertise. In the realm of software development several methods have been developed - Structured Design [63], Jackson [20], and others. A method confers
general rules, guidelines, suggestions, and standards to follow in solving a problem in the domain of application. The person applying the method must provide the creative details. Not enough is known or understood to automate the solution. But the method can structure the solution and lead the human problem solver to consider all the necessary options. A software environment which supports an appropriate methods extends a greater degree of human engineering. This method can also aid in tool integration by providing a framework for analysis of the project information. (The TRIAD environment provides method support by coding methods in an attributed grammar form [34, 1] that underlies and structures the project information.) In a field or domain about which little is known, in which experience is limited, or which is too complex and encompassing, the semantic dissonance may to great to overcome. In such a domain, methods of problem solving are under-developed or nonexistent.

5.2.3. The View Extractor and Output Distributor

The task of overcoming syntactic dissonance can be accomplished by means of a view extractor in combination with an output distributor. Syntactic dissonance results from a use of a particular syntax, granularity, or level of cohesion in the information base and another in the tool. A view extractor is a generic system component which extracts the necessary information from the information base, changes the syntax to that expected by the tool, and then submits that data to the tool as input. A corresponding component, the output distributor performs the reverse operation. The results of the tool are distributed to the spot in the information base that best reflects the context of the those results. The tools results in context are more meaningful and more easily understood.
These two components enable the existing tools to be incorporated into a system with no internal changes to the tools. A view extractor and output distributor are written to provide the correct view for the tool and to distribute tools results. The task of building these two components is much simpler, in general, than constructing a new tool. This is especially true for sophisticated tools to be used for critical operations. Since the tool is unchanged, its reliability is not subject to question. The construction of the view extractor and the output distributor is made especially easy if the information base provides commands or functions to traverse and query the project information. Two such components are necessary for each tool which is to be integrated in as much as the view extractor and output distributor must be aware of the syntax requirements of the tools and the structure of the information base. (The development of a generator for the view extractor and/or the output distributor is theoretically feasible, and is a possible area of future research and development.)

5.2.4. The Model

An important feature of most programming environments which incorporate syntax-directed editing is that the common information base is structured by means of a model, usually a grammar for the particular language upon whose syntax the system is built. This underlying model provides not only a means of organizing and retrieving the data, but also a framework in which to analysis, synthesis, and inheritance of information can take place. The presence of a similar model of the project information can have corresponding benefits for a more
encompassing software environment such as TRIAD. In particular, the information frames model discussed in section has proven to be an especially useful one. The presence of the model, while not imperative, presents a useful mechanism for the efficient and effective integration of existing tools. It is possible to use other models, for example the relational model, to represent and structure the project information. However, the information frames model, implemented via attributed grammar forms [34], has several inherent advantages for tool integration.

The presence of the grammar is vital to the view extractor and the output distributor. The task of the view extractor is divided between a global controller and local action routines. The knowledge of the grammar enables the global control to easily traverse the project information in an order which reflects its grammatical organization. The grammar is reflected in the presence of logical groupings of project information. A local action routine can then be associated with each of these logical chunks. An understanding of the semantics of the chunk along with a knowledge of the syntax necessary to fulfill the view required by the tool permits the local action routines to pad the data of each chunk with the appropriate syntax to satisfy the tool's expectations.

The TRIAD software environment is supported by such an information model for these and other reasons [34, 46]. The question to be addressed here is whether such a model is necessary and/or beneficial to the task of the integration of existing tools.

The use of a grammar form model as opposed to a grammar model is advantageous for conceptual reasons that do not directly pertain to tool integration. These include the fact that a grammar form allows instantiated project information to be reflected at all symbols of the grammar, not only at the terminal ones as is true of traditional grammars. Furthermore, attributed grammars permit the tuning [34] of a grammar. The software engineering method which is modeled by the attribute grammar form is a dynamic entity. It evolves as more experience and expertise is gained in a particular application domain. It is vital to allow these methods to evolve, or to be tuned, to reflect the current experience and standards. A grammar form model allows this tuning to take place in a more dynamic manner.
Another construct which bears upon the nature of the information base in a more conceptual manner is the association of qualifying attributes with logical components of the system. (These attributes may be represented physically as another field in a relation in a relational database.) Such an association of attributes with local data adds considerable meaning to that data. The context of its appearance contributes to its semantics. This use of attributes originates in data flow analysis [19] techniques where the attributes are "use" and "define" qualifiers that describe the use and definition of variables in a section of code. The concept can be usefully extended to other levels of project information by means of attributes to depict ownership, completion level, version information, and a myriad of other adjective phrases.

These attributes have proven useful in tool integration primarily as a matter of efficiency. The recording of information in its context and the automatic propagation of that information in an inherited or synthesized manner, furnishes accurate, timely semantic data about the project information in an immediately accessible fashion. The same information could be deduced by a detailed analysis of the project information, although this long, algorithmic process would have to be performed each time the data was required. The attributes and associated semantic functions act in a more incremental manner to continually update this information. (This is directly analogous to the operation of incremental tools, and has similar benefits.)

The implementation of this model is another question altogether. Factors to be considered in the choice of implementation strategy include the speed and efficiency of retrieval, ease of use for the user, types of queries permitted (anticipated versus deductive or unanticipated), etc.
One implementation technique, the use of a relational database, is discussed in the next section.

5.2.5. The Project Information Repository

Storing information in, and retrieving it from, the information base is a primary task of a software environment. The project information is the most important resource that a corporation or organization has. A database of some variety is necessary to fulfill the storage and retrieval needs. An existing database can provide the requisite functionality if it allows project information to be stored and retrieved as needed. The query mechanism of many existing systems provides a facility for answering the unanticipated type of question that often arises in project development. A database management system (DBMS) is designed to handle a large volume of information such as will be produced by project members and by tools. Some aspects of project development lend themselves more readily to DBMS storage than do others. The structure and relationship information may be voluminous, yet so non-repetitive that the DBMS cannot be used to its full advantage. For example, although there may be many "while" loops in a piece of code stored in the information base, there may be only one document called "Requirements for Phase II". The duplication of record types is a technique that is exploited by database management systems to efficiently process large volumes of data. The single document mentioned above could be stored with other documents in a documents record, but the fact that this is a "Requirements" document would not be contained in the record type. It could be recorded as a separate field in the record, but not as part of the structuring information of the project information. Any analysis of the information base which was founded on the structure
of the project information would not be aware of this "Requirements" description.

5.3. Useful Techniques for Integration of Existing Tools

Just as the concepts and constructs discussed above are useful in integrating existing tools into a software environment, the following techniques are useful in the actual implementation of such constructs.

In designing a software environment and, in particular, in developing a view extractor or output distributor, the separation of global from local control seems apropos [24]. The global control for a view extractor permits the specification of the order of traversal of the structural project information that is required for extraction of the requisite data. Local action routines, as used in Gandalf [18, 17] and TRIAD [26, 27, 28, 46, 59], can then be used to implement the local aspects of the extractor. These local action routines can be set to automatically fire upon entry to, exist from, or modification of a logical chunk of project information. The global control can invoke these local action routines when it visits each portion of information.

This separation makes modification of the code much easier. The global control has been decoupled from the local context. The functionality has been further distributed by use of these local actions. This decoupling has the effect of lowering the number of interactions for which to account in developing these components. A change in the information requirements of the tool which is reflected in only a specific, isolated portion of the project information will consequently require a change in one specific action routine. A change in the structure of the information base will require changes in the global control without affecting the individual action routines.
A second technique that has proven useful is the use of attributes and associated semantic functions. An attribute is a descriptive element associated with a particular logical chunk of the project information. Storing of information in its context makes it more valuable and useful. These attributes may include names of variables defined or used in this chunk of project code, the owner of a chunk, the cost of the project component represented by this chunk, etc. (The same data can be represented globally if a description of the local context is included. But this entails a duplication of that contextual information with the added problems of wasted resources and consistency.) In the TRIAD system, these attributes are an integral part of the model underlying the project information. This model is represented by an attributed grammar form.

Each attribute can have associated with it a semantic function which describes some procedure for propagating the value of this attribute to neighboring chunks of project information. This propagation of local information from one chunk to the next and from there to a third, etc. allows the accumulated local information to have a global effect. As alluded to previously, data flow analysis using attributes [] is an example of this type of propagation of information.

A technique which has been described previously is the use of existing interactive capabilities to modify the behaviour of a tool. By using the times when a tool pauses for user input, the monitor can regain control from the tool. Having obtained control, the monitor can invoke other tools, retrieve information from the information base, provide an improved prompt for the user, etc. Furthermore, the monitor can simulate the incremental operation of a tool by means of this technique. By switching control from tool to tool, the monitor can present the impression to the user that the tools are operating incrementally.
The trapping of output of a tool to redirect to another location is necessary if existing tools are to cooperate in an environment. The results need to be stored in the information base rather than in an external file in order that other tools may use the data produced. In a complementary sense, the tool must allow the monitor and, indirectly, the user to control the tool and to integrate it into the system without rewriting or modifying the tool in any manner. (The Unix operating system on which one version of TRIAD is built provides the pipe mechanism and input output redirection [15] which can be used to accomplish these tasks.)

Since syntactic dissonance is one of the major hurdles to cross in integrating existing tools, syntax padding to achieve syntactic harmony is a vital technique. This padding occurs in two methods. Local syntax, i.e. context free syntax, can be provided by the action routines. These action routines are associated with local chunks of logically related information, and are associated with a particular tool. Hence, the knowledge of the tool's syntactic requirements and an understanding of the structure of the project information is available to these routines.

Syntax additions and corrections of a more global nature can be made by examining the remainder of the project information to find the needed referents. This is obviously not the task of a local routine, but needs to be accomplished by a more global agent similar to the global controller. This again illustrates the need for and the advantages of a separation of control into local and global components.
5.4. Advantages of this Approach to Integration

The integration of existing tools into a software environment by the techniques herein described has been depicted as ergonomic, effective, and economic. The primary ergonomic benefits can be summarized in the statement:

The user's focus of attention remains on the project information rather than on the tools or other system components.

To be more precise, these benefits include:

1. a more uniform tool interface,
2. automatic invocation of some tools,
3. tool operation and cooperation in the background,
4. more incremental tool operation,
5. presence of a monitor to smooth (or obscure) the tool transitions,
6. padding of syntax to remove syntactic details from the user, and
7. improved command syntax.

Each of these can enhance the user's productivity by maintaining
his/her concentration on the problem being solved of the system component being designed, while removing non-productive distractions and details.

The effectiveness of the integration of existing tools is demonstrated by the increase in the power of the tool commands that are possible by means of tool cooperation and tool monitoring. (Specific and detailed examples of these enhance tool commands are explicated in the following section.) This increased effectiveness is gained by means of:

- ability of the monitor to capture run-time status,

- the use of the monitor in conjunction with the query mechanism of the project information base,

- the use of the monitor as a common interface to the tools,

- the monitor coordinating the application of tools on the project information base,

- the grammar model for the monitor and the information base,

- the ergonomic presentation of the results of the tools, and

- the grammar and the accompanying attributes.

The increase in existing tool effectiveness by means of integration is more apparent when the capabilities of such a system are contrasted with other systems as in figure 11. Not only can the power of simple tools be multiplied, it can often surpass that of more sophisticated and complicated systems.
All of these advantages would be rendered impotent if the price, in terms of design and coding time and complexity, was much greater than for the creation of new, more integrated tools. The concepts and techniques introduced in this dissertation are an attempt to simplify this task of integration. Their success has been demonstrated in the accompanying implementation, i.e. the integration of an existing compiler and symbolic debugger into the TRIAD environment. A comparison of the amount of code produced and the effort to produce it between systems of similar (in fact less) power, and the corresponding measures for the integration of the compiler and debugger establishes the utility and economy of these techniques and concepts. (See Figure 12.) One is forced to the conclusion that time and effort spent in the design and implementation of new tools which are not conceptually different or more powerful than existing tools is an ineffectual use of these resources. These resources would be meaningfully applied to the integration of the existing tools in a way that increases their user friendliness and power of these tools.

5.5. The Human-Engineering Potential of Harnessing Existing Tools

To demonstrate the efficacy of the harness, this section will examine ways in which it can be used to achieve an ergonomic software environment. Sample commands are given to illustrate various points.

Commands facilitated by

the ability of the monitor to capture run-time status:
<table>
<thead>
<tr>
<th>System</th>
<th>Number of Lines of Code</th>
<th>Time Spent in man/months (if available)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPS</td>
<td>15,00 to 20,000 of C</td>
<td>12</td>
</tr>
<tr>
<td>Pecan</td>
<td>12,000 of C</td>
<td>18</td>
</tr>
<tr>
<td>Magpie</td>
<td>40,000 Pascal 2000 C and assembler</td>
<td></td>
</tr>
<tr>
<td>Gandalf</td>
<td>47,000 of C</td>
<td></td>
</tr>
<tr>
<td>Triad</td>
<td>4,500 of C</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 12: Comparisons of Some Measures of Effort

As an interface between the user and the tools and the project information base, the monitor can easily determine the user's current focus of attention in the information base from the cursor position. From this information, and from a knowledge of the expectations of the tool, the monitor can create the correct user command to reflect the user's context (i.e. command-syntax padding). The monitor can also bind a single command to commonly used sequences of tool commands. Since the system architecture places the monitor as a controller over the other tools, a single command to the monitor can easily be reflected as a sequence of tool commands to several different tools. As the monitor develops and as the project’s domain becomes better understood, the monitor can become more of an expert system. At that point, the monitor can provide abbreviated commands for frequently used sequences, on its own, by observing usage patterns.
Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Tool</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Breakpoint</td>
<td>Debugger</td>
<td>Monitor examines the context, i.e. the cursor position, to determine the location at which to break, then passes this information (with the correct syntax) on to the debugging tool.</td>
</tr>
<tr>
<td>Find errors</td>
<td>Editor, static analyzer, data flow analyzer</td>
<td>First invokes the static analyzer, then the data flow analyzer. The cursor for the editor is then positioned at the first error.</td>
</tr>
</tbody>
</table>

the use of the monitor in conjunction with the query mechanism of the project information base:

A flexible query mechanism under control of the monitor permits the development of more powerful tool commands. The monitor can initiate queries on the project information base. The information extracted by the query can be used to augment and control the commands given to the tool. Since the information base query mechanism is assumed to be flexible enough to handle unanticipated user queries, the commands based on this facility are quite dynamic in nature. These commands are not statically developed at system development or system generation as the typical tool commands. Rather, these environment commands use the information base to modify and control the operation of the tools.
Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Tool</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set a breakpoint at every statement that alters the variable &quot;sum&quot;.</td>
<td>Debugger and database query tool</td>
<td>Use the query mechanism to retrieve all the statements that alter &quot;sum&quot;, then pass the statement location on to the debugger.</td>
</tr>
<tr>
<td>For every project that is behind schedule, send a message to its manager</td>
<td>Mail system and database is query tool</td>
<td>Use the query mechanism to retrieve the names of the managers of all projects that are behind schedule. Then send a message via the mail system to each of these.</td>
</tr>
</tbody>
</table>

the use of the monitor as a common interface to the tools:

The monitor presents a common interface to the tools. The change of modes each time a different tool is invoked is not necessary. The user is aware only of the project information - not the idiosyncrasies of each tool.

The monitor must also be flexible in its binding of commands to keys and must permit the evolution of new combinations of commands. This evolution should preferably be permitted to occur dynamically rather than to require a recompilation of the system. Many table driven system, e.g. the EMACS editor '60', provide this ability.
The user can benefit from not having to remember and re-input parameters from one tool invocation or user session to another [41]. These parameters include the last operation performed by the tool, the position of the cursor when last used (more generally, the chunk of project information that was the focus of attention when this tool was last invoked), the last sequence of tool commands and parameters, the last state of the current chunk of project information (e.g. compiled but not linked), or the last error produced by application of the tool. The monitor is ideally suited for remembering global parameters. Systems such as Argus [64] and Toolpack [41] provide methods for maintaining this global type of information.

Most existing systems lack the facilities to retain local information in its context. Parameters that pertain exclusively to individual portions of the information base are more efficiently and effectively stored with that local chunk. More specifically, attributes retain historical information for individual chunks, which, if stored in a global attribute, is lost. For example, the cost of developing an individual module should be saved with that module as an attribute. Conversely, if the cost of individual modules are accumulated into a global attribute called "overall cost", information pertaining to individual modules will be lost unless explicitly maintained.

The local attributes are also useful for storing local tool status information from invocation to invocation. For example each logical portion of the text may have a cursor position attribute associated with it to record the last position of the editor tool when visiting that chunk.

the monitor coordinating the application of tools on the project information base:
Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Tool</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug &quot;program1&quot;</td>
<td>Debugger and information base</td>
<td>Access the information base to find the last input to the debugging tool. Apply that input script to the debugger.</td>
</tr>
<tr>
<td></td>
<td>base query mechanism</td>
<td></td>
</tr>
<tr>
<td>Locate the site of the last</td>
<td>Interpreter and editor</td>
<td>Recover the last error from the information base, then run the program stopping at that location</td>
</tr>
<tr>
<td>error produced when the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>current program was last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interpreted.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Who was the last to modify</td>
<td>Editor and local attributes</td>
<td>Access the &quot;modifier&quot; attribute of the current node and display the result.</td>
</tr>
<tr>
<td>the current chunk of project</td>
<td></td>
<td></td>
</tr>
<tr>
<td>information.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tool commands can be made more powerful by the coordination of several tools. The monitor can automatically call a tool or a sequence of tools when it detects some condition. Since the user interface is monitoring the project development process, it is located in precisely the appropriate spot to accomplish this.

the grammar model for the monitor and the information base:

The grammar that underlies the project information structurally relates the chunks of project information. This relationship information has the following advantages:
Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Tool</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>After every 10 steps of the editor, invoke the compiler on the partially developed program.</td>
<td>Editor and compiler</td>
<td>The monitor counts the number of steps of the editor, and when 10 are finished calls the compiler.</td>
</tr>
<tr>
<td>Protect this section of project information by informing the manager.</td>
<td>Editor and mail system</td>
<td>Each time that an unauthorized user tries to change this section of the project information send a message to the manager of this project.</td>
</tr>
<tr>
<td>Freeze this version of the procedure (i.e. form a rudimentary version control system).</td>
<td>Editor</td>
<td>Protect this procedure from changes by not passing the commands from the interface to the editor when the monitor perceives that the commands are attempting to change a frozen procedure.</td>
</tr>
</tbody>
</table>

- enables the tools to be method-driven.

- allows the retrieval of analytical information that can be used to guide the tools.

- allows the efficient recovery of the user's context.
allows the query mechanism to answer unanticipated, deductive queries rather than merely to retrieve recorded project, and

allows the monitor to pad the syntax of the project information (e.g. program code) before sending it to tools.

Examples

<table>
<thead>
<tr>
<th>Commands</th>
<th>Tool</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send the unfinished program to the compiler to test the parts that are completed.</td>
<td>Compiler and editor</td>
<td>The code is collected from the project information base. Its syntax is padded from knowledge provided by the underlying grammar.</td>
</tr>
<tr>
<td>Enforce the company's programming standards</td>
<td>Editor</td>
<td>The programmer is guided through process of programming by the grammar which encode the company's standards. The editor is driven by the syntax of this grammar.</td>
</tr>
</tbody>
</table>

the ergonomic presentation of the results of tools:

The information resulting form a tool application can be presented in a more human-engineered form by using the monitor, the output distributor, and the view extractor. The monitor records the level of
detail of interest to the user. (The user can, of course, modify this level.) The view extractor provides the appropriate level of summary or detail information. The output distributor is responsible for placing appropriate fragments of the results back into the project information base along with the chunks that generated those result fragments.

Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Tool</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step to next statement</td>
<td>Debugger and output</td>
<td>Invoke the debugger, trap its output, map this back into the project</td>
</tr>
<tr>
<td></td>
<td>distributor</td>
<td>information, then reflect this in the position of the cursor.</td>
</tr>
<tr>
<td>Compile the current program</td>
<td>Compiler, editor, and</td>
<td>Extract the code from the information base and send it to the</td>
</tr>
<tr>
<td>and find the errors in the</td>
<td>view extractor</td>
<td>compiler. Then insert the error messages back into the information</td>
</tr>
<tr>
<td>project information base.</td>
<td></td>
<td>base along with the corresponding code.</td>
</tr>
</tbody>
</table>

the grammar and accompanying attributes:

Attributes become more valuable if allowed to affect changes in their immediate neighborhood rather than being strictly limited to a single chunk of the information base. The inclusion of action routines to
Propagate relevant information to neighbors allows local information to eventually have some non-local effects. Selective instrumentation of the code and the application of the tool to these pieces of the code become a possibility [45].
Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Tool</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test a particular procedure.</td>
<td>Editor and compiler</td>
<td>Access the local &quot;used&quot; and &quot;defined&quot; attributes to determine which identifiers needed to be defined to permit the compilation of this procedure. The monitor can obtain these from other portions of the information base by use of these attributes or else from the user. The user is prompted for a driving routine.</td>
</tr>
<tr>
<td>Which section of project was accessed most frequently.</td>
<td>Editor</td>
<td>Access the &quot;read&quot; and &quot;write&quot; attributes of each section of the project information to determine the most frequently accessed.</td>
</tr>
<tr>
<td>Are there any potential errors in the control structure of the current procedure?</td>
<td>Editor and attributes</td>
<td>Access the &quot;define&quot; attributes of the control structure of each loop and compare with the &quot;define&quot; attribute of that loop body. If any are used in the control but are not modified in the body, notify the user.</td>
</tr>
</tbody>
</table>
Chapter 6
Implementation

The actual implementation, to validate the above approach to tool integration, was to integrate a C compiler and symbolic debugger into a grammar-based TRIAD harness. TRIAD is a software environment that supports the process of software development. The underlying project information base component is hierarchically organized. The model underlying this information base is an extended attributed grammar form [7, 22, 29] which is also used to encode a project method presented by the interface. This method can be changed and adapted to a particular project by changing the underlying grammar. This model is the basis for the operation of many of the TRIAD tools. For example, the syntax-based editor is customized by a specific method. This editor differs from typical syntax-directed editors because the underlying grammar is not restricted to a programming language. Consequently, the TRIAD system can support the entire life cycle of software development.

The user is actually presented with blank forms in which to fill chunks of project information. These forms are refined by other more specific forms. These forms are actually user-friendly presentations of the productions of the underlying grammar and, thus, the forms are related by the underlying productions in a systematic way. The result of this process is a forest of trees of forms that contain the project information in a hierarchically-organized canonical format.
The process of integrating tools into the TRIAD environment consists of three steps:

- designing the forms from which the input information for the tool can be easily extracted and into which information can be inserted if necessary,

- specifying a global controller to traverse the filled forms in the appropriate order, and

- coding of local action routines, to be invoked by the global controller, to extract the information from the associated chunks in the forms.

6.1. Design of the Forms

A goal in the design of forms is to guide the user during project development and elicit just the right information. Another goal is to maintain the information needed by tools that are to be integrated into the system. The design of the forms can affect the difficulty of writing the global control and the local action routines necessary for this integration.

The example presented in figure 13 is fairly simple but, nevertheless, illustrates the potential for uniform support from design to coding. The coding forms reflect the syntax of the C programming language. (See Figure 13.) The form headings present the various C constructs and relieve the user of much of the syntax details.
Functional Description: This program is a simple demonstration program which finds the average score on a test.

Revision History [More?] :

Revision Number: 1 | Date: 11-5-84 | Changed By: J. Kiper

Change: Simplified program for demonstration.

Rationale: No one is interested in figuring out an involved program when learning how a new system works. This is not a test of every feature of the system, but a demonstration of the overall method.

External Data Definitions [More?] : none

Include Files: none

Procedure Name and Argument List:

Name: main

Type Returned - if any:

Input Arguments: [More?] none

Output Arguments: [More?] none

(13,14 Procedures Used [More?] : ) none

(15 Procedure Body:) Form-use-#[ 107 ]

See form # 107
Local Data Definition [More?]:

Name: i
Type: int
Description: A loop variable

Local Data Definition [More?]:

Name: test[100]
Type: float
Description: Used for the class test scores

Local Data Definition [More?]:

Name: ave
Type: float
Description: Used for the class test average.

Local Data Definition [More?]:

Name: sum
Type: float
Description: Used to accumulate the total of the test scores.

Procedure: (Continued on next page.)
Procedure: Position cursor at appropriate statement type in menu. Enter "Z*R. Then position cursor at correct position in code and enter "Z*H to insert the new statement.

Menu
- While Loop
- For Loop
- Input/Output
- Assignment
- procedure call

I/O Statement: Form-use-# [ 109 ]
Input: scanf("%d", n)
Output:

Assignment Statement: Form-use-# [ 110 ]
Object: i
Source expression: 0

While Loop: Form-use-# [ 111 ]
Condition: i <= n
Body:

I/O Statement: Form-use-# [ 109 ]
Input: scanf("%d",test[i])
Output:

Assignment Statement: Form-use-# [ 115 ]
Object: sum
Source expression: sum + test[i];

Assignment Statement: Form-use-# [ 116 ]
Object: i
Source expression: i - 1

Assignment Statement: Form-use-# [ 112 ]
Object: ave
Source expression: sum / n

Figure 13: Forms Containing a C Program
6.2. View Extractor for the C Compiler

The view extractor for the C compiler was implemented in C. Its design also demonstrates the effectiveness of separating the overall logic into global control and local actions.

6.2.1. Global Control for the View Extractor

The global control is responsible for visiting the nodes of a form tree containing a C language program in the appropriate order. Figure 14 gives the pseudo-code which describes the operation of the global control for the view extractor.

6.2.2. Local Action Routines for the View Extractor

The local action routines associated with each node of the underlying parse tree are responsible for collecting the C code from the forms, padding the syntax correctly, and inserting it into the output file. Figure 16 also contains pseudo-code for some of the local action routines.

The local action routines are also responsible for collecting some additional data which is used by the output distributor. This is considered a view of the information base needed by the output distributor. This data is a correlation between the line number of each statement in the output file and an identification of the position in the form tree node from which it was taken.

As the global control (see Figure 14) traverses the entire tree and triggers each local action routine in the appropriate order, the C program chunk has been extracted from the form tree and inserted in the correct linear order in the output file. Figure 17 illustrates the source program
begin
node := Get'Next'Node ( heading == 'Program Name' );
if node != NULL'NODE then
  Action'Routine'1 ( node );
node := Get'Next'Node ( heading == 'Local Data Definition' );
while node != NULL'NODE do
begin
  next'node := Get'Next'Node ( node );
  Action'Routine'2 ( node, next'node );
  node := Get'Next'Node ( heading == 'Local Data Definition' );
end
node := Get'Next'Node ( heading == 'Procedure'Body' )
form := Get'Child'Form ( node );
if form != NULL'FORM then
begin
  Visit'Form ( form );
  node := Get'Next'Node ( heading == 'Statement' );
  while node != NULL'NODE
begin
    form2 := Get'Child'Form ( node );
    node2 := Get'Next'Node;
    Statement'Routine ( node2 );
  end
end
end.

Statement'Routine ( node )

case of node - heading
  'While Loop' : Action'Routine'3;
  'For Loop' : Action'Routine'1;
  'Assignment' : Action'Routine'5;
  'Input/Output' : Action'Routine'6;
  'Procedure Call' : Action'Routine'7;
end;
end;

Figure 14: The Global Control for the View Extractor
Procedure: Position cursor at appropriate statement type in menu. Enter "Z"R. Then position cursor at correct position in code and enter "Z"H to insert the new statement.

Figure 15: The Errors Inserted in the Forms by the Output Distributor
Action_Routine_1 ( node ) { For the program name }

begin
  print ( output_file ( node );
  print ( output_file, ';' );
end;

Action_Routine_2 ( node1, node2 )
{ For the local data definitions }

begin
  Print_Text_in_Node ( node2 ); { output the type }
  Print_Text_in_Node ( node1 ); { output the name }
  print ( output_file, ';' );
end;

Action_Routine_3 ( node1, node2 );

begin
  print ( output_file, 'while' );
  Print_Text_in_Node ( node1 );
  { output the while condition }
  print ( output_file, '\n(' );
  form := Get_Child_Form;
  if form != NULL_FORM then
    begin
      node := Get_Next_Node ( heading == 'Statement' );
      Statement_Routine ( node ); { for body of the loop }
    end
  print ( output_file, '\n})\n' );
end;

Figure 16: Some of the Local Action Routines for the View Extractor

that results from the application of the view extractor to the forms of figures 13A and B. This file is then presented to the standard C compiler for compilation.
main
{
    int i;
    float test[100];
    float ave;
    float sum;

    scanf ( "%d", &n );
    i = 0;
    while ( i < n )
    {
        scanf ( "%d", &test[i] );
        sum = sum + test[i];
        i = i + 1;
    }

    ave = sum / n;
    printf ( "%s %5.2f", "The class average is", ave );
}

Figure 17: The Output File Produced

The separation of global control from local actions makes it easy to adapt the view extractor to changes in the project development method. If changes are made to the productions underlying a form, only the action routines associated with that form have to be changed. If new productions are introduced into the grammar (and consequently new forms are produced), the global control is modified to include the new
forms and new action routines are created for the new forms. But the other action routines are unchanged. This decoupling has effectively lowered the complexity by lowering the number of interactions among the various action routines.

6.2.3. Attributes Use by the View Extractor

A more sophisticated view extractor collects the information from the global context that is necessary for the testing a particular procedure or incompeleted program. The primary environmental requirement is that the use and maintenance of local attributes be supported in some method [7, 22]. In particular, this task requires the use of two attributes associated with each node of the tree (i.e. with each production of the grammar). One attribute reflects the identifiers that are used in the subtree rooted at the node, and another reflects the identifiers that are defined in the subtree. Then, to permit the testing of an incomplete program, the values of these attributes at the root node are examined to determined which identifiers are used but are not defined. If the identifier is a procedure or function name, the system can insert a stub of appropriate type. If the identifier is a constant or variable, a temporary declaration of the correct type (as determined from the context of its use) is inserted. The user is permitted to submit his her own temporary or permanent variable declarations or procedure function definitions for these. The completed view is then extracted and compilation is commenced.

If a particular function or procedure is to be tested outside of the main program (as is necessary in a modular development method), the attributes of the node corresponding to that procedure are examined to ascertain which identifiers are used but are not defined in this subtree.
The remainder of the tree outside of this subtree (i.e. the complement of the subtree) is then systematically searched to find these definitions. (This search can efficiently proceed by examining the attributes of the ancestors of the procedure's root node from that node toward the root of the tree until an ancestor node is located which lists the identifier in question as defined. Then, the "defined" attribute of its children are examined to find the one which lists that identifier as defined in its subtree. This search is continued until the appropriate definition is located.) This definition is then included in the extended view.

6.3. Output Distributor for the C Compiler

The output distributor takes as input the correlation data produced by the view extractor plus the error statements produced by the C compiler. The error statements from this particular C compiler consists of the line number at which the error was identified and a brief description of the error. Using the correlation information, the output distributor places the error statement back in the form tree immediately after the offending statement. Figure 15 shows some of the same forms as in figure 13 with the errors inserted at the appropriate places by the output distributor.

6.4. Debugger Integration

The primary view for the debugger is the core image created when the compiled program is run. The only "view" that the environment needs to create is the correct syntax for commands. The TRIAD interface acts as a monitor to retrieve contextual information in order to make the commands more useful. To set a breakpoint, for example, the user positions the cursor at the point in the program at which he/she wishes to suspend execution and then issues the command "set breakpoint".
During execution of the program, when a breakpoint is reached, the user can position the cursor at a particular variable and issue the print value command. From the form chunk that contains the declaration associated with that variable, the monitor can collect the type information necessary to give the correct command to the debugger to print the value of the variable.

The output distributor for the debugger has the responsibility of changing the cursor position to reflect the operation of the debugger. The DBX debugger ordinarily outputs a message like "breakpoint at line 143 in proc'namc" to indicate that a breakpoint has been reached. The output distributor converts this to the action of positioning the cursor at that line in the form tree. The user can then, at a glance, comprehend the context. For example, the user can not only examine the statement at which the break occurred, but also can check surrounding statements and data declarations.

When the user issues a "print value" command, DBX ordinarily outputs the value. The output distributor places the value in a new, adjacent window. It also highlights in the program the variable whose value is being printed. Thus, the context can be seen in one window and the value in the adjacent. The same window is used to collect other subsequent data. It serves as a record of the debugging session which can be useful for later analysis.

The TRIAD system provides a flexible query mechanism. This can be used, for example, to identify all occurrences of the assignment of a new value to a particular variable. The monitor can then use this information to set breakpoints at each of these locations.
6.5. Commands

In the TRIAD system, the number of possible useful commands is virtually unbounded. The compiler and the debugger subsystems are similarly very fertile. The ability to modify commands by means of data obtained from the project information base adds another factor to the increase in commands. Obviously, all the possible commands cannot be implemented. A set of "primitive" commands was chosen that could provide an adequate base for manipulating the compiler and the debugger while demonstrating the feasibility of the implementation techniques and the power, and user friendliness of the resultant integration. The discussion of this subset of primitive commands which follows is organized by purpose - compiling, interface checking, debugging, and code instrumenting. A set of "composite" commands is then discussed. These commands are a composition of queries to recover information from the project information with tool commands modified by this information.

6.5.1. Primitive Commands

The basic compiler command is one which extracts a complete program from a form tree, adds the necessary syntactic information, and sends the resulting file to the compiler. This command is supplemented by another which positions the cursor at the first error in the program. (Since "first" is somewhat ambiguous with respect to a tree, this is interpreted to mean the error which corresponds to the statement with the lowest line number in the extracted file.) Subsequent application of this command will move the cursor to the position of the next error.

The compile command at the procedure level tests the syntax of a particular procedure. Since the C compiler which is being used
change is not incremental, compiling an entire program repeatedly can be quite time consuming. By allowing the syntax of one procedure to be checked, response time can be shortened. To permit the testing of one procedure at a time, the user is prompted for a driving routine. This serves as the main program for this procedure, and is compiled with the procedure. The "find first error" command used after compiling a single procedure causes the cursor to be positioned at the location of the first error in that procedure.

The C compiler can also be used to check the syntax of an individual statement. A simple main program is automatically composed which includes this statement and all other definitions active in the scope of that statement. (This is, of course, more definitions than are needed. When a lexical becomes available, it could be used to determine the presence of variables, procedure or function calls, or other identifiers. The list of definitions to be included could then by efficiently reduced to an optimal number.)

Interface checking is the process of determining if all procedure calls are consistent with their definitions. In C this includes determining that the type of actual parameters and formal parameters is compatible. This check cannot, by the nature of typing in the C language, be very complete. Coercion of types is permitted and often used. (This is a feature of this language which adds flexibility, but whose misuse and over use produces abstruse code.) This technique is demonstrated in the context of C although it would be more useful in a more strongly typed language like Pascal or Ada. The implementation technique is to collect all the code that involves the definition or use of a procedure. This includes the procedure name, the formal parameter list, the declaration of
the types of these parameters, and a simple procedure body that contains only procedure calls and the definitions of the actual parameters of this call.

When applied at the procedure level, the same operation is performed on all code in the subtree rooted at that procedure. By submitting these procedure calls and definitions to the compiler, any mismatches not allowed in C are detected.

The method of integrating the DBX debugger permits all the commands of that tool to be used. Other capabilities and improved command syntax adds more powerful commands and increased user friendliness for all commands. In addition to these improvements and additions, all the original commands are permitted in their previous syntax.

The most used commands of the debugger are:

- Set breakpoint,
- Print the value of a variable.
- Execute the next statement (i.e. step), and
- Change the value of a variable.

Various versions of these important commands are provided by this integration whose syntax and usage is adapted to make them more palatable to the user. Each of these commands can, when appropriate, be applied with different results at several levels - at the statement level, the control structure level, and the procedural level.
A breakpoint can be set at a specific statement in the program by positioning the cursor at that statement in the tree of project information, then issuing a simple command. (A breakpoint is a notation in the program to cause the execution to temporarily pause at that precise point in the program.) The simple command is generic in that no identification of the location of the statement has to be made. This information is taken from the context of the user's focus of attention (as indicated by the position of the cursor.)

A particularly practical and prevalent debugging operation is the setting of a breakpoint just prior to, inside, and immediately after a control structure. This is accomplished in this implementation via a single command. The control structure of application is determined by the position of the cursor. The control structure used is that one which immediately encloses the statement at the current cursor position. A similar flat sets breakpoints in the procedure in which the cursor lies.

The printing of the value of a variable is logically only at the statement level. The user positions his/her cursor at the declaration of the variable in question. The monitor can extract the name of the variable and use this to compose the command that is necessary for the DBX debugger. The user is protected from the error prone task of correctly typing the name of the variable in the precise command syntax. When a lexical analyzer become available, this capability can be extended to the use of the variable name in any statement where it occurs. With such an extension, the user positions the cursor on the variable name in any statement. The lexical analyzer is then used to determine the variable name. This name is, in turn, used by the monitor to compose the appropriate DBX command.
A primary capability of any debugger is to control the execution of a program by stepping through the program one statement (or a few statements) at a time. Stepping at the statement level is provided by DBX. The monitor can adjust this step size by issuing a sequence of "step" commands before returning control to the user. A more useful technique is to allow the user to step over an entire control structure. Thus the monitor can step over a "while", "repeat" or "for" loop to avoid the monotony of stepping through a large number of iterations of the loop.

The ability to change the value of a variable is similar to the process of printing the value of a variable. The user's cursor is positioned at the declaration of the variable. The new value is entered by the user in response to a prompt from the user, and is used by the monitor along with the variable's name to issue the precise DBX command.

As a part of the testing and debugging process, it is useful to instrument the code. This is the process of monitoring the execution of a portion of a program by furnishing the user with data (feedback) about the state of the execution environment. At a minimum, this data is a dynamic report about which procedure, statement, or control structure is currently being executed.

Despite the fact that this instrumentation is part of the debugging process, its implementation was achieved without using any symbolic debugger. In response to a command to instrument an individual statement, the monitor inserts an identifying print statement immediately before and after that statement in the program before submission to the compiler. At the control structure level, the result is the insertion of the appropriate print statements before, inside, and after that structure. The
procedure level is handled in a parallel manner. When the program is run, either alone or by the debugger, these print statements help the user to monitor the progress of the execution of the program. This additional data may help identify the possible areas of error to be further explored with the symbolic debugger.

6.5.2. Composite Commands

The composite commands are those whose implementation is not explicitly supported by either the C compiler or the DBX debugger, but require intervention and processing in a significant manner by the monitor. (This categorization is subjective and is used only as an organizational technique to present the commands.) The first group of composite commands to be discussed can be further characterized as static. Once the command is formed, there is no additional processing necessary from the monitor during execution of the command. The general form of these commands is "Execute the tool command on the portion or portions of the information base which satisfy the query". The monitor determines the location of the one or more logical portions of the form tree which satisfies the query. This data is then used by the monitor to compose the necessary tool commands. The usable queries, at this point in the development of the system, have to be anticipated and their solution determined when the system is compiled. (With the introduction into the system of a more adequate database management system with a sophisticated query language, this facility can be made much more flexible and, hence, more powerful.)

Some examples of these commands are:

- Compile all procedures which contain a reference to the variable "x".
• Check the interfaces of all procedures which were modified since February 1.

• Find the next syntax error caused by missing parenthesis.

• Set a breakpoint at all statements which reference the variable "sum".

• Set a breakpoint at all procedures which were modified since January 15.

• Print the value of all variables used within the current control structure.

Dynamic composite commands are those in which there is interaction between the monitor and the tool while the command is being executed. These commands generally have the form "Execute the command while the dynamic condition is true". Examples of these dynamic, composite commands follow:

• Compile the current program after each 100 editor commands.

• Compile the program when editing is finished on procedure "input'scores"

• Step to the next statement while \( x > 0 \).

• Stop when a procedure is called which was modified since February 15.

In summary, these commands are a small sample of the total possible.
This subset does demonstrate the potential gain in power and user friendliness obtained from an application of these concepts and techniques.
Chapter 7
Summary and Conclusions

7.1. Summary

7.1.1. Motivation: The Advantages of the Integration of Existing Tools

The primary motivations for this research center on the improved use of existing software tools by their integration into a software environment. Tools are integrated when they communicate and share with each other (often through a common project information repository). By being more cognizant of the overall purposes of the project development environment and being aware of the presence of other tools, an integrated tool can more effectively aid in the development process. The set of tools for an environment, when integrated, become a team rather than a group of self-centered individuals.

Tools in cooperation are often more powerful than the aggregate potential of the tools in isolation. The whole, in this instance, is greater than the sum of its parts. (This synergism is demonstrated by the combination commands possible in the integration of a C compiler and a symbolic debugger.)

Not only the power, but also the user friendliness of the tools is enhanced by an integration of tools. The common interface to the tools (i.e. the monitor concept) permits a uniform syntax for the use of tools in the tool kit. This greater potential for human engineering arises from
the fact that the interface/monitor allows the programmer to concentrate on the most important entity in the organization - the project information. His/her attention is not fragmented by the need to invoke tools, produce the tool's input in the appropriate syntax, wait for the tool to complete its action, and finally returning the output produced to the correct context for analysis.

Integrated tools that operate in an incremental (or simulated incremental) manner can ergonomically improve the environment by virtue of a faster response time and more timely user feedback. An incremental tool is able to operate in small steps by the re-use of work accomplished on previous invocations. Since only a small amount of work needs to be accomplished for each application of the tool, the response time to the user is faster. Furthermore, the tool can be used more frequently resulting in more accurate, up-to-date information. The cycle of manipulation of the project information and tool feedback in reaction to that development can occur more frequently. Errors or problems in a portion of the project information can be identified while the user's attention is still focused on that chunk.

Many contemporary software development tools have evolved to such a sophisticated level that there remains little resemblance to their ancestors of a few years ago. This evolution will continue as new tools evolve and developing concepts become practical. To be able to solve the difficult problems and develop the complex systems demanded of modern software development organization, this evolution and development of tools must proceed. Conversely, software tools are a major item in the budget of any such organization, both in initial cost and maintenance. Wise stewardship of organizational resources mandates effective and long term
use of these investments. Integration of existing tools in which the organization has previously invested can provide a viable alternative to the development or purchase of the latest software tool. By increasing the power of the set of tools and increasing their usability, the tool investment can be protected and prolonged.

The wisdom of such an extension to the life of existing tools is magnified by consideration of the user’s level of confidence in these software aids. Existing tools with a long record of reliable use instill a high level of user confidence that should not be hastily discarded. A detailed knowledge of the characteristics and idiosyncrasies of a tool can only be obtained through experience. Newly developed tools have no established reputation of reliability and often do not deserve one. The number of errors or oversights in early versions of software are often sufficient to warrant extreme caution in their use. Existing tools have overcome these problems (or else the users have become aware and compensate). Thus, the integration of existing tools provides a feasible method of protecting and better utilizing an organization’s software investment.

7.1.2. Characterization of Tool Integration

This research provides a characterization of levels of tool integration by enumerating parameters of this integration. These parameters are:

- granularity,

- cohesion, and

- harmony/dissonance
Each of these terms has several related meanings when applied to various components of the software environment. **Granularity** refers to the size of the accessible units of the component, and ranges from coarse to fine. The granularity of the information base is the size of the logical chunks of project information to which the user or tools have access. The granularity of tool interaction describes the length of the time interval between user interactions with the tool. Analogous definitions hold for the term granularity as it relates to the interface, tool control, etc. In general, a finer granularity produces an environment in which tool integration can more easily take place.

**Cohesion** describes the amount of structure which is present among the various grains. With reference to the information base, cohesion specifies the amount of structural information present to relate one logical chunk of project information to another. The cohesion of a tool's incremental operation prescribes the degree of sharing that occurs between incremental invocations of the tool. Cohesion has similar meanings when applied to other system components.

The **harmony** (or dissonance) of some quality in two or more components is the degree to which that quality is shared (or not shared) among those components. Two tools with harmony of project information granularity agree in the size of the unit of their access to the project information which they use or produce. Harmony of tool incrementality means that two or more tools agree in their degree of incremental operation. It is clear that harmony is to be preferred in most situations.

These three parameters, granularity, cohesion, and harmony/dissonance, when taken together, can characterize the degree of tool integration present among a group of tools of in an entire system. These can also be
used to predict how successfully a tool can be integrated into an environment.

7.1.3. Useful Techniques

Having given a parameterized characterization of the levels of integration, this research proceeded by an enumeration and discussion of techniques which are useful for achieving this integration among existing tools. These techniques include the following:

- a method

- a monitor

- a view extractor

- an output distributor

- a database

- attributes and semantic functions

A method is a set of rules or standards to be followed by project members in the task of program development. When this method is described in some rigorous manner (as in the attributed grammar forms of TRIAD), the project information base can be structured by this model. The tools can then use this model to facilitate the retrieval and analysis of information in the database.

A monitor is an extended user interface which serves as a controlling agent and record keeper over the tools. The use of a monitoring agent
can produce more uniform interactions between the tools and the user. The varying syntax of tools can be somewhat standardized for the user through the technique of syntax padding. More importantly, the monitor allows the user to focus his/her attention on the project information while automatically calling the necessary tools at the appropriate times. The monitor can help to achieve interaction between tools by passing information between them to modify their input commands.

A view extractor is requisite given the philosophy at existing tools should be used without change. The task of this component is to extract the necessary information from the project information repository and to pad the syntax to produce that which is expected by the tool. An output distributor performs the inverse task by inserting the results of tools into the project information in the most appropriate context.

The project information base is the cornerstone around which any environment is built. This information repository can most appropriately be implemented by means of an existing database management system. The query language facility of many of these systems provides a useful means of implementing the many unanticipated queries needed to answer questions for the user (i.e. manager, programmer, analyst, etc.) and to provide data for the tools.

Attributes allow the recording of local information in its context in a format which can be modified and manipulated by local semantic functions. These functions can be written easily as their interactions with other portions of the project information are limited to neighboring chunks. When project information in their domain of influence is modified, semantic functions can propagate these changes to related portions of the database.
7.1.4. Demonstration of Feasibility

The climax of this research is a demonstration of the feasibility of the concepts and techniques enumerated. This takes the form of an integration of an existing C programming language compiler and a symbolic debugger into the TRIAD software environment. The underlying philosophy was to use these tools without modification. The result was an integrated environment which allows the use of the full power of both tools. In fact, their usability and power are increased. The increased usability arises through a more uniform and user friendly interface provided by the monitor. The increased power comes from the sharing of information and control among the monitor, the tools, and the information base. A complete set of commands was designed and an assortment implemented to demonstrate the important techniques and concepts.

7.2. Conclusions - What Has Been Learned

This research has served as an existence proof of the various concepts and techniques. The implementation, in particular, has fulfilled this purpose by demonstrating in a tangible way the pragmatism of

- loose levels of integration,

- the integration of existing tools,

  - demonstration of the ease of this integration,

  - demonstration of the increase in user friendliness, and

  - demonstration of the power of synergism.
• the prescribed integration techniques, and

• incremental (or simulated incremental) tools.

In each case, this implementation serves as a proof by example of the utility of these, not a proof of their necessity or sufficiency for an adequate software environment.

7.2.1. Loose Levels of Integration

Many contemporary systems have demonstrated the advantages to be gained from a tight integration of tools (for example, the Cornell Program Synthesizer [67]). This integration of a C compiler and a symbolic debugger into the TRIAD software environment shows similar advantages in a loose integration. The integration is looser in that either the tools share a lower volume of information or a less detailed granularity of information, or else the sharing occurs less frequently. The tools are not tightly coupled together. This has economic advantages especially in terms of the cost of construction of the tools and the price of their integration. When such a loose integration has most of the advantages of tight integration, as in this implementation, the cost-utility tradeoff favors the lower (i.e. looser) level of integration over a tighter, more costly integration. There seem to be some instances in which a looser integration can give the user a better sense of control over the tools and the environment, and therefore actually be more usable.
7.2.2. Integration of Existing Tools

One of the primary conclusions of this research is that the integration of existing tools has at least three distinct advantages over the creation of new integrated tools. The first is again an economic consideration. The use of existing tools eliminates the commitment of human and machine resources necessary to develop, or the large expenditure of company funds required to purchase, sophisticated tools. The software development necessary to achieve this integration into TRIAD was relatively minor when contrasted with that of building or buying new tools. Furthermore, the learning phase in the use of new tools is eliminated in existing tools.

The second benefit of the integration of existing tools is the increased user friendliness of their use. This research has presented demonstrative proofs of the greater usability of the tools when integrated. Techniques for accomplishing this improved engineering were enumerated. The primary conceptual discovery and guiding principle was that the user's focus of attention should not be distracted from the project information. Tools should perform their very important tasks as unobtrusively as possible.

Finally, the power of synergism generated by tools in loose cooperation was demonstrated. The size of the set of additional commands possible when tools share information is substantial. These are commands that are not possible for the tools in isolation without extensive user intervention.

Another conclusion of this work is that the prescribed techniques for integrating existing tools can, indeed, accomplish that task. Their existence and established feasibility provide the groundwork for their application to other environments. (Some of these extensions are discussed in section to follow.)
This work has brought to light the value of (simulated) incremental tool operation. This concept is being explored and exploited with great success in compiler theory and construction. The techniques presented here can simulate this mode of operation in many compilers which were not designed with incremental operation in mind. Furthermore, these mechanisms are not restricted in their application to compilers.

Incremental operation is advantageous to the user primarily because of the rapid and more timely receipt of feedback from the tools. The feedback can be more rapid because the tool is able to re-use work previously accomplished to avoid repetition. It can be more timely since a tool that operates quickly can be invoked more often. The result is an environment in which the user is better able to concentrate on the project information since feedback on errors and other information about a logical chunk of project information is received while the user's attention still resides on that chunk.

7.3. Future Research Directions

The research discussed in this dissertation has prompted many more questions to be answered and directions to pursue. These can conveniently be grouped into implementation extensions and more theoretical concepts.

7.3.1. Implementation Extensions

The most obvious extension to the current implementation is the integration of other tools into the environment. The most interesting to attempt to integrate would be those oriented to programming-in-the-large of programming-in-the-many. This would allow the prescribed techniques to be tested under different conditions and in a wider domain.
The possibilities and limitations in the use of attributes and semantic functions has not been fully explored. A view extractor may possibly be very efficiently and simply coded by means of semantic functions acting as attributes under the control of some more global agent. New tools themselves may function well when their functionality is distributed among these semantic functions.

The current implementation uses a rather ad hoc scheme to store the project information on a secondary storage device between TRIAD sessions. Much could be gained from the integration of an existing database management system. Not only would this provide a more organized and efficient mechanism for the storage of project information (especially as the volume grows), but the query mechanism of many such systems would furnish a method answering unanticipated queries. This would be useful to users in recovering needed project information. In addition, the tools and the monitor could use such a mechanism to search the database for information to modify or to combine commands.

A more ambitious implementation extension would be the application of the described techniques to other software environments. Even though most are missing several key ingredients of the support provided by the TRIAD environment (primarily the method model underlying the project information), many techniques remain to achieve some level of integration of existing tools in these environments. The ultimate success of these mechanisms as applied to other software environments remains to be explored.
7.3.2. More Conceptual and Theoretical Extensions

In the current implementation, a view extractor and output distributor must be built for each tool to be integrated. A generator for the view extractor or for the output distributor would overcome the primary difficulty in the integration of existing tools. Given the knowledge of the structure of the information base (i.e. the attributed grammar form in the case of TRIAD) and the syntax of the tools' input (for the view extractor) or the tools' output (for the output distributor), a generator is conceptually possible. A problem lies in the fact that many tools have used no grammar or other model to describe the syntax of its input or output.

More precise comparative studies of the usability and effectiveness of integrated versus unintegrated tools would establish with greater reliability the value of integrating existing tools. This would involve a more psychological style of experimentation with human subjects. The benefits would likely vary when novice users are contrasted with experts.

The use of incremental tools to increase the user friendliness and usability of tools needs to be further explored. The following questions should be investigated:

- What characterizes a reasonable level of incremental operation?
- Is incremental operation useful for tools other than compilers?
- What typifies tools which can be built to operate incrementally?

This survey of possible future directions is by no means exhaustive.
These represent some of the extensions currently of greatest interest. Pursuit of any one of these will indubitably expose other related topics.
Bibliography


28. Kuo, J. C.  *Design and Implementation of a Form-Based Software Environment*. Ph.D. Th., The Ohio State University, Columbus, Ohio, August 1983.


58. Shubra, C. J. Jr. The Identification of Semantics For the File/DataBase Problem Domain and Their Use in Template-Based Environment. Ph.D. Th., The Ohio State University, Columbus, Ohio, August 1984.


61. Standish, T. A. A Preliminary Philosophy for ARCTURUS. Computer Science Department, University of California, Irvine, California, 1980.


