Techniques for the Integration of Existing Tools

James Kiper
Miami University, commons-admin@lib.muohio.edu

TECHNICAL REPORT: MU-SEAS-CSA-1988-003

Techniques for the Integration of Existing Tools

James D. Kiper

Techniques for the Integration of Existing Tools

by

James D. Kiper
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #88-003 03/88

1843E

Techniques for the Integration of Existing Tools

James D. Kiper, Ph.D.

Systems Analysis Department

Miami University

Oxford, Ohio 45056

March 1988

Techniques for the Integration of Existing Tools

SUMMARY

The purpose of this paper is to explain and demonstrate the advantages of tool integration and the reuse of existing tools. Several techniques for the integration of existing tools are presented and discussed. These techniques include the use of a monitor, simulated incremental operation, syntax padding, view extraction, and output distribution. The advantages of the use of these methods of tool intregation are illustrated by their use in integrating an existing compiler and on-line debugger. The commands that are produced by this synergism have increased both the user friendliness of the tools and the power of the resultant commands.

Key words:

Integration, software tools, incremental operation, techniques of integration, software development, software environment

INTRODUCTION

The primary objective for the research described in this paper is to improve the quality 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". This tool integration concept has been a theme of many recent programming environments [4,8,12,14,18,20].

Benefits of Tool Integration

The conceptual benefits of tool integration are dual. First, tools in cooperation have a synergistic effect which 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 may 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 added efficiency, combination commands, which are more powerful than single commands to individual tools, can be developed. (Examples of such cooperation and the possible commands will be enumerated in later sections.)

The second conceptual advantage of integrated tools is an increased level of user friendliness. This is notably demonstrated in a 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.

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. The use of existing tools in such a cooperative environment has additional advantages, not the least of which are the 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 has been demonstrated by means of an implementation. (This integration of an existing compiler and a symbolic debugger into a software environment will be discussed later in the section on implementation.)

A vital issue is the user's level of confidence in a tool. Although a new tool is a technological advancement, initially it may not be perceived as such by the user. Newly produced tools often evolve through several versions in which improvements are made and serious errors are corrected. In addition, user productivity normally declines until the idiosyncracies of a new tool are learned; analogous idiosyncracies in an older tool have been mastered. Furthermore, 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. 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 user's faith in the reliability and usability of the overall system. An environment which can incorporate existing tools has increased flexibility and adaptability. It is not locked into the syntax, structure, or capabilities of the current set of tools. Furthermore, the potential exists to develop commands for these existing tools that are even more sophisticated than those performed by more technologically advanced tools.

Related Work

The value of integrating tools has been widely recognized in the computer science and software engineering communities over the past decade. Many tool collections are now designed with a view toward integration. The most successful of these are the language sensitive editors which are effectively a very tight integration of an editor and a translator (compiler).

In addition to this work, there is an increasing awareness of the need to integrate existing tools. At the University of Illinois, the Illinois Software Engineering Program has proposed an open systems architecture whose goal is to make the addition of new tools simple [9]. This is to be accomplished through a "tool bus" or "software backplane." This group proposes the development of as many intercommunication protocols as necessary, rather than forcing all tool communication to occur through a common protocol. The "partial sort morphisms" of their work are the view extractor and output distributor of this research. (These terms, view extractor and output distributor, will be discussed subsequently.)

Work at the Software Productivity Consortium has centered on the development of a framework into which new and existing tools can be integrated [1, 15]. This framework hinges on a project library, a session coordinator, and a "harness." The session coordinator is responsible for dynamically composing tools as requested by the user, orchestrating the execution of these tools, and assuring that the data needed by tools are available. The project library is a collection of technical and project management data and relationships among this data. A "harness" is a piece of software which mediates the differences in the data requirements of the tool and those of the data base. The research described in this paper has determined an analogous division of labor.

Another approach to the problem of tool integration was taken by Chalfan at Boeing [2, 3]. In this work, information about the relationships among various tools was incorporated in the rule base of an expert system. Given a designed output, this expert system was designed to check for the inputs which would produce that output. If those inputs were present, that tool was invoked. If those inputs were not present, the expert system's rules are evaluated to determine which tools (if any) could be invoked to create those inputs. (The recursive nature of this approach easily can be seen.) The significance of this work is in the capturing of the relationships among tools in the rule base for an expert system. This technique worked well in the given application area (design of aerospace vehicles) because the tools involved generally produced simple, numeric data rather than the highly structured data typical in software development environments.

These ideas have begun to appear to a limited extent in commerically available software. A prime example of this software is the Language Sensitive Editor developed by Digital Equipment Corporation for its Vax family of computers [22]. This product works with all the programming languages for which DEC furnishes Vax compilers. Thus, existing compilers can be integrated with an editor. (The implementation of this tool is not clear from the user manual. The syntactic knowledge about these languages seems to have been coded into this tool. However, it is clearly possible to invoke the compiler from inside the tool and to have the results of the compile available to the editor.)

TECHNIQUES FOR INTEGRATION

This section will describe some techniques which have proven useful in the task of integrating existing tools. The topic of degrees and categories of tool integration are discussed in a companion paper [11]. Figure 1 illustrates the relationship among the primary components of such an integration.

The Use of a Common 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. (This is analogous to the "session controller" in the work at the Software Productivity Consortium, SPC [1,15].)

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.

Tool as Child Process

The advantages of a common monitor or user interface to the set of tools is magnified if the tools can function as a child process of the monitor. The potential to operate in such a manner is dependent upon the operating environment in which the implementation takes place. If the operating system allows concurrent processes, and if the tool to be integrated uses the facilities provided by that operating system, then that tool may be able to function as a child process of the monitor. This allows the monitor to exercise a greater amount of control and coordination over the various tools.

The Use of Input Pauses

The monitor can conveniently exercise control over child process tools by the effective use of times at which the tools pause for user input. (It is possible for the monitor to obtain control from the tool at arbitrary points during the tool's execution. This would require modification to the tool's code. Our purpose here is to avoid such modification because of the inherent complexity of that task in general.)

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 and storing each tool's state information, the monitor can present the impression to the user that the tools are operating incrementally.

When the tool pauses for user input, the monitor can usurp control and save the state of the tool in the project information base. 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 [5,6,16] by providing information from multiple sources within a small interval of time.

The monitor can incorporate the information base 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 using a database tool to determine the locations of all statements that modify the variable "x". This can then be used to send several commands to another tool, the debugger, which sets breakpoints at each of these locations. This task is beyond the scope of either tool working individually without user intervention.

The Use of a 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 inanimate resource that a corporation or an 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 database 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 that produced by project members and by tools.

In descriptions of the integration work at the SPC, the project information repository is referred to as the "project library" [1,15]. In the framework used in the related work at Illinois, a database of project information is not a primary component. They recognized the need that most software development projects have for a database by including it as a tool [9]. Chalfan's work at Boeing introduces the possibility of storing tool relationship data in an information base [2,3]. (This work does not center on software development. Consequently, storage of project information is not discussed.)

If such a project information base does not exist, the first step in the integration of existing tools is to develop one. The ability to gather all the project information into a central location has not only the obvious benefits to project management but also aids in the integration of tools. These integration advantages are further explicated in subsequent sections.

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. 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 those results. These results are more meaningful and more easily understood in context. (See Figure 2.)

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 tool 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 since 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.)

The need for the view extractor and output distributor is echoed in the SPC integration work in their discussion of a "harness" for integrating existing tools [1,15]. At the Illinois Software Engineering Program, these conversion tools are given the appellation "sort monphisms" [9].

Trapping of Output in the Information Base

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 provides the pipe mechanism and input/output redirection [7,21] which can be used to accomplish these tasks.)

Syntax Padding

Since syntactic dissonance, i.e. incompatibility in the syntax of tools, is one of the major hurtles to cross in integrating existing tools, syntax padding to achieve syntactic harmony is a vital technique. (Syntactic dissonance is further explicated and illustrated in [11].) This padding occurs in two methods. Local syntax, i.e. context free syntax, can be provided by "action routines." These action routines are procedural components which are associated with local chunks of logically related information, and are associated with a particular tool [19]. 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.

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 term "user-friendly" has been used so often it has become almost meaningless. To be more specific, the primary benefits which are ergonomic , i.e. user-friendly, 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 the following:

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

Each of these can enhance the user's productivity by maintaining his/her concentration on the problem being solved or 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 enhanced tool commands are explicated in the following section.) This increased effectiveness is gained by the following:

- o the ability of the monitor to capture run-time status,
- o the use of the monitor in conjunction with the query mechanism of the project information base,
- o the use of the monitor as a common interface to the tools,
- o the coordination by the monitor of the application of tools on the project information base, and
- o the ergonomic (user-friendly) presentation of the results of the tools.

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 paper are an attempt to simplify this task of integration. Their success has been demonstrated in an accompanying implementation, i.e. the integration of an existing compiler and symbolic debugger. Although accurate data is generally not available, some approximate size and development time data from developers has substantiated the intuitive hypothesis that integration of existing tools involves less work than development of new tools with similar power. 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 more productively applied to the integration of the existing tools in a way that increases their friendliness and power.

AN INTEGRATION OF EXISTING TOOLS

The power of the integration of existing tools in general, and the aforementioned techniques in particular, is illustrated best by a discussion of an actual integration. The tools chosen for integration were the standard C compiler and the DBX on-line debugger provided by Unix. The TRIAD [13,17] software environment provided the project information base and the monitor. (See Figure 3)

In such a 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.

Primitive Commands

The basic compiler command is one which invokes the view extractor to obtain a complete program from the project information base, adds the necessary syntactic information, and sends the resulting file to the compiler. The program is stored in the information base as a tree. The information store at each node of the tree includes much more than program instructions. Documentation and various descriptive attributes are also stored at each node.

Upon completion of the compilation, the output distributor places the errors back into the tree with the statement at which the error occurred. A command is provided to position 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 without 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 which is produced by the view extractor. 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 view extractor automatically composes a simple main program 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 separate lexical analyzer 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 be 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 nature of typing in the C [10] 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 for a view extractor 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. An output distributor can then convey this information to the user in the most effective manner.

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:

- o Set breakpoint,
- o Print the value of a variable,
- o Execute the next statement (i.e. step), and
- o 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, when appropriate, can 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) by a view extractor.

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 used is that one which immediately encloses the statement at the current cursor position. A similar fiat sets breakpoints in the procedure in which the cursor lies.

The printing of the value of a variable logically makes sense at the statement level only. The user positions his/her cursor at the declaration of the variable in question. A view extractor can extract the name of the variable and can 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. An output distributor is responsible for obtaining output from the debugger tool and presenting it to the user.

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 name of the variable 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 the minimum, this data is a dynamic report about which procedure, statement, or control structure is currently being executed.

Despite this instrumentation being 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 be- fore, 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.

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 and a view extractor. (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 characterized further 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 tree which satisfies the query. This data then is packaged by a view extractor 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 the following:

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

CONCLUSIONS

This paper has presented some techniques for the integration of existing tools. The need for such integration is two-fold. First, integration of tools is a paradigm which has proven quite effective in improving user friendliness and the power of tools. Secondly, the integration of existing tools can bring the power of integration to set of tools which have been proven reliable, with which users are comfortable, and in which there has already been a large financial investment.

The effectiveness of these techniques has been demonstrated in the integration of an existing compiler and debugger. The commands discussed in this paper are new commands which neither tool was able to provide by itself. In general, the interactions with the user for these new tools are more understandable and user friendly than the commands and messages of the original tools.

In the final analysis, such integration has been demonstrated to be a cost effective way of improving the quality of the tools currently available.

REFERENCES

1. "Build 1 Demonstrates Harness Technology for Off-the-Shelf Tools." The Software Productivity Consortium Quarterly, Vol.2, No. 1, January 1988.

2. Chalfan, Kathryn M. "A Generic Tool for Integrating Software Components." The Boeing Company, 1987.

3. Chalfan, Kathryn M. "A Knowledge System that Integrates Homogeneous Software for a Design Application." The AI Magazine, Summer 1986.

4. Donzeau-Gouge, V., G. Huet, G. Kahn, and B. Lang. "Programming Environments Based on Structured Editors: the Mentor Experience." Inria, May 1980.

5. Feiler, P. H. and R. Medina-Mora. "An Incremental Programming Environment." Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, April 1980.

6. Fritzson, P. "Preliminary Experience from the DICE System a Distributed Incremental Compiling Environment." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, PA, April 1984, pp. 113-123.

7. Gillette, D. "UNIX Time-Sharing System." The Bell System Technical Journal 57, 6, Part 2, July-August 1978, all.

8. Habermann, A. N. and D. S. Notkin. "The Gandalf Software Development Environment." Department of Computer Science, Carnegie-Mellon, Pittsburgh, PA, 1982.

9. Kaplan, Simon, et al. "An Architecture for Tool Integration." Proceedings of the Workshop on Advanced Programming Environments, Torndhiem, Norway, June 1986.

10. Kernigham, B. W. and D. M. Ritchie. The C Programming Language. Prentice-Hall, Englewood Cliff, N J, 1978.

11. Kiper, James D. "The Integration of Software Development Tools." Technical Report 87-001, Miami University, April 1987.

12. Osterweil, Leon J. "Toolpack - An Experimental Software Development Research Project." IEEE Transaction on Software Engineering 9, 6, November 1983, pp. 673-685.

13. Ramanathan, J. and D. Soni. "Design and Implementation of an Adaptable Software Environment." Computer Languages 8, 3/4, 1983, pp. 139-159.

14. Reiss, S. P. "PECAN: Program Development Systems That Supports Multiple Views." Seventh International Conference on Software Engineering, IEEE Computer Society, Orlando, FL, March 1984, pp. 324-333.

15. Riddle, William E. "Future Software Engineering Environments." Keynote Address, ACM Computer Science Conference, Atlanta, GA, February 24, 1988.

16. Schwartz, M. D., N. M. Delisle, and V. S. Begwani. "Incremental Compilation in Magpie." Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM, Montreal, Canada, June 1984, pp. 122-131.

17. Soni, D. A. Design and Modeling of TRIAD - An Adaptable Integrated Software Environment. Ph.D., The Ohio State University, Columbus, OH, June 1983.

18. Standish, T. A. and R. N. Taylor. "Arcturus: a Prototype Advanced Ada Programming Environment." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, PA, April 1984, pp. 57-64.

19. Stonebraker, Michael, Jeff Anton, and Eric Hanson. "Extending a Database System With Procedures." ACM Transactions on Database Systems, 12, 3 (September 1987), 350-376.

20. Teitelbaum, T., T. W. Reps, and S. Hortwitz. "The Why and Wherefore of the Cornell Program Synthesizer." SIGPLAN Notices 16, 6, June 1981, pp. 8-16.

21. Unix Programmer's Manual. Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkely, CA, 1984.

22. Vax Language-Sensitive Editor and Vax Source Code Analyzer Guide, Digital Equipment Corporation, Maynard, Mass., August 1987.

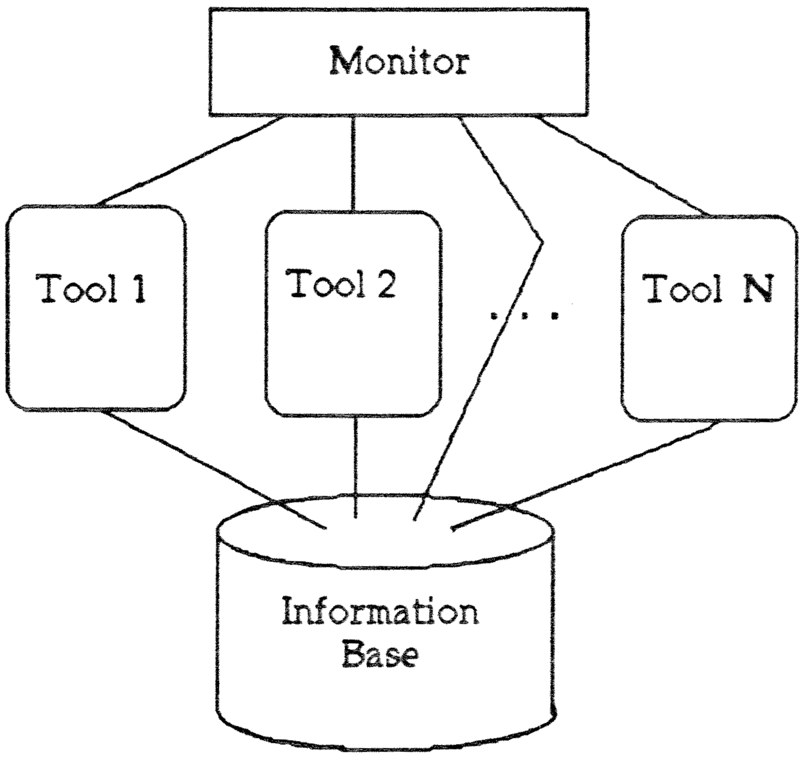

Figure 1: System Structure for Integration of Existing Tools.

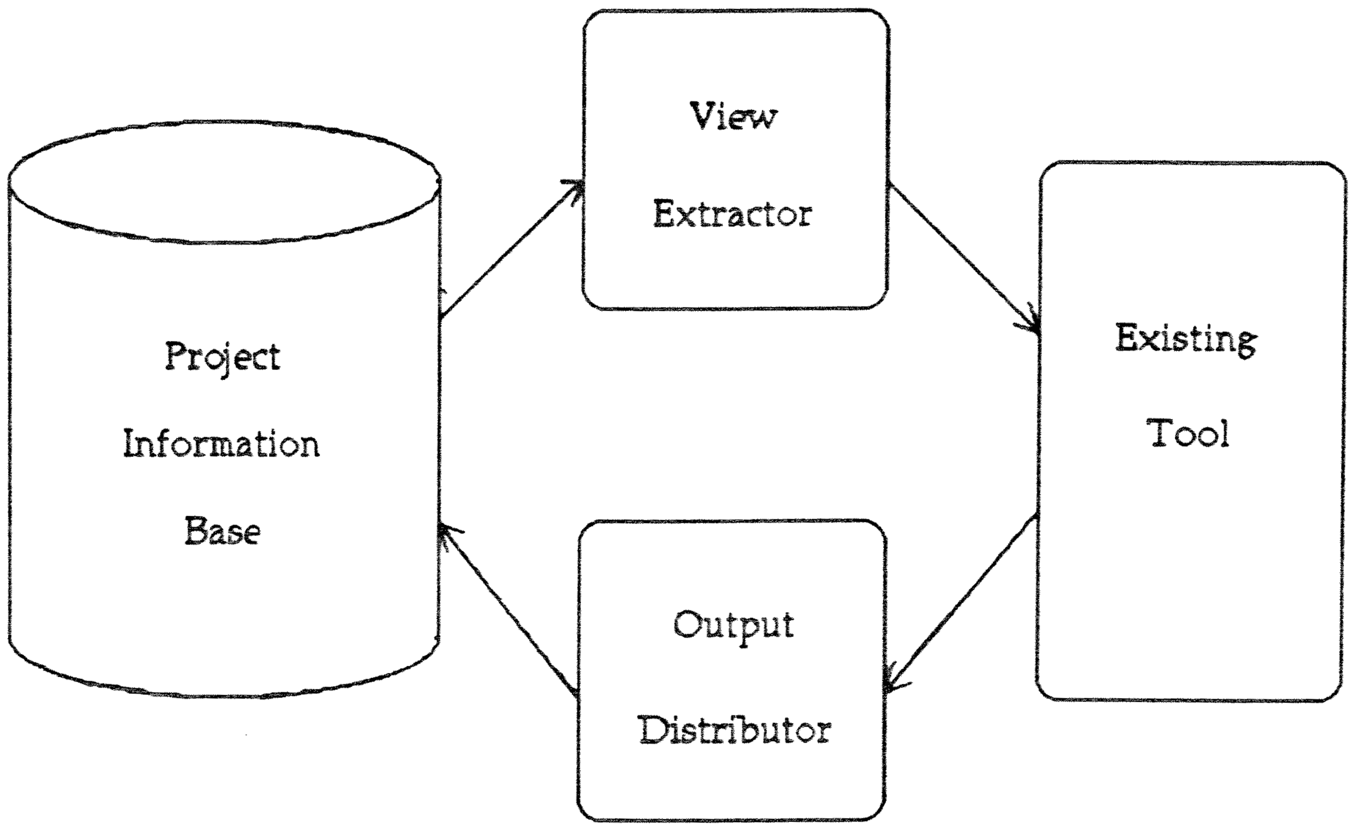

Figure 2: View Extractor and Output Distributor

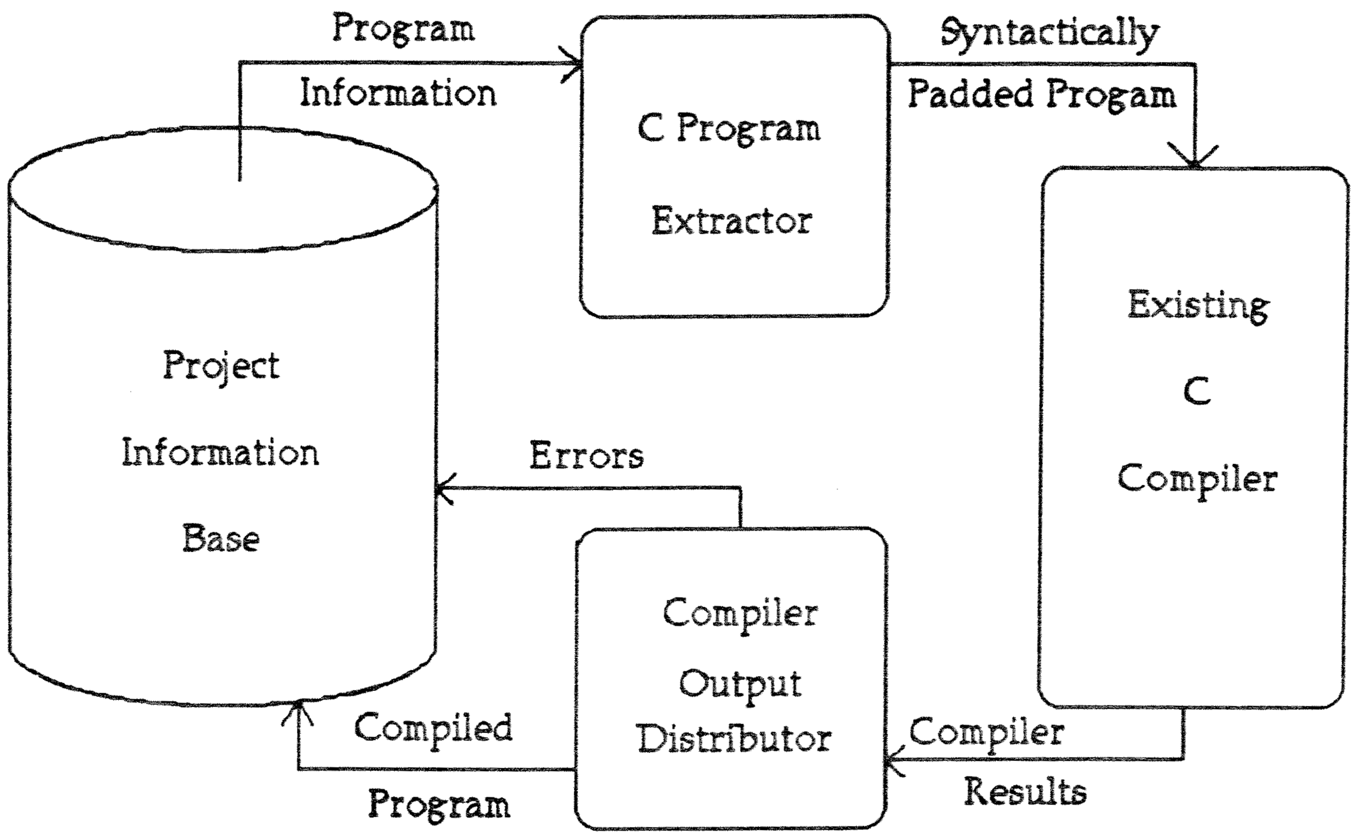

Figure 3: Structure of the C Program Extractor and Output Distributor