The Integration of Software Development Tools

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

TECHNICAL REPORT: MU-SEAS-CSA-1987-001

The Integration of Software Development Tools

James D. Kiper

The Integration of Software Development Tools

by

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

Working Paper #87-001 04/87

Index Terms--Software tools, integration, software environment, incremental tools, integration categorization, project information base, case interface.

The Integration of Software Development Tools

0. Abstract

The effectiveness of software development tools can be dramatically increased by their integration (i.e. their cooperation). This paper discusses the problems to be overcome in integration of tools, and a categorization of the degree of tool integration. The continuum from loose to tight integration is parameterized. An informal method is described to apply these parameters to tools in order to determine some measure of their ability to be integrated.

1. Introduction and Problem Definition

Software tools of many types have proven their usefulness and even their indispensability to software development over the past thirty years. A tool, in the context of software development, is a software component which aids the user in the performance of tasks of software construction. This assistance varies from actually accomplishing those tasks that can be automated, to giving advice or providing data. These tools enable more effective and efficient use of computer and human resources permitting the user to concentrate on the more creative aspects of project development while allocating the more mundane tasks to the tool. These tool-automated tasks are often performed with greater accuracy, reliability, and speed than is possible by human effort alone. Consequently, the effective work accomplished by the confluent efforts of a user and software tools is several orders of magnitude greater than that of the user in isolation. (The comparison of automated compilation versus compilation by hand or the use of machine language is an obvious illustration of this gain.) This paper will examine the tool integration problem, the benefits of integration, and some parameters for the characterization of levels of tool integration. It will conclude by presenting an informal method for evaluating the extent to which existing software tools can be integrated.

2. Bridging the Gap

Software tools have been continually developed and improved as additional portions of the software development task are automated. The progress has not kept pace with the rate of increase in the complexity and size of software projects. The resulting "software crisis" has been well documented. This continual need for more powerful tools can be met in two ways. The most obvious method is the development of new, more powerful, more usable tools with increased functionality. The other solution is the improvement of the power and usability of existing tools through their integration. The second method is the topic of this paper.

The development of new, improved tools ultimately represents the ideal solution to the software crisis. However, the technological transfer time necessary to incorporate a major conceptual improvement into a commercially available and accepted tool can take as long as twenty years and the immediacy of the problems mandates a swifter solution [14]. The integration of existing tools can provide an acceptable alternative in reducing this technological transfer time. If integration can be accomplished in a cost-effective manner, it has the added benefit of improved stewardship of an organization's software investment.

2.1 The Tool Integration Problem

One of the primary conceptual limitations of most existing software development tools is a lack of cooperation with other tools. (See figure 1.) This lack of cooperation often effectuates a duplication of effort since tools performing different but related tasks, share the need and, therefore, duplicate

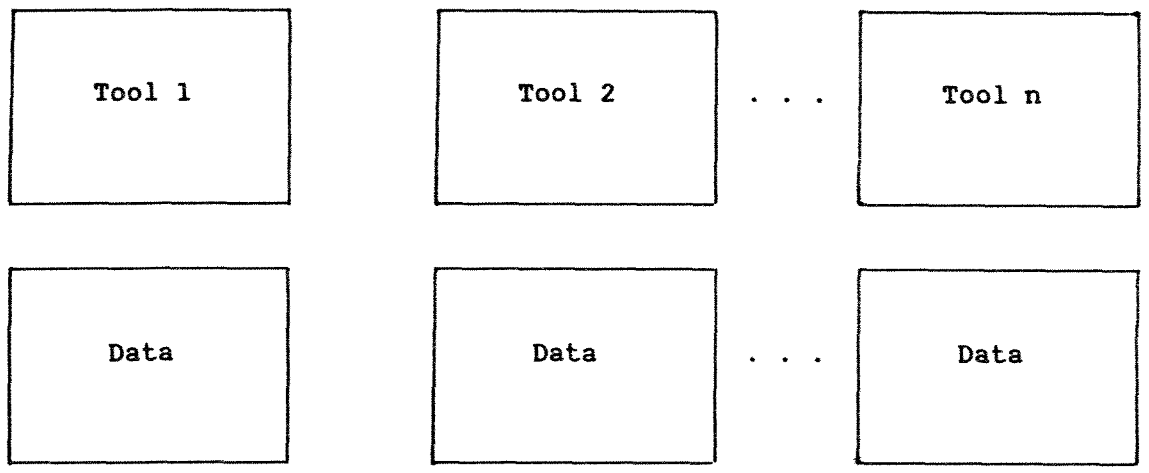

Figure 1: Un-Integrated Tools Lack Cooperation

some of the same steps. For example, an accounting package may produce a file of records. Another tool is to subsequently sort this output by record number. 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 application. For example, 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 many symbol table entries, tokens parsed, and even sections of code produced are identical from one invocation of the compiler to the next.

This 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 solely to facilitate its own task. The input to one tool which arises from the output of another tool requires a translator to produce compatibility. Conversely, if tools were developed with the optimization of the entire software development process as a goal, the user would ultimately be better served. This inevitably would require some tools to perform added processing to conform to the standards of the system. One such standard that has often proven pragmatic is a common information representation for use by all tools in their input and output. One simple standard for tool communication, the character stream, has been used by the Unix^{TM} system to achieve a certain level of tool cooperation.

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. Consequently, the user, particularly the novice, is faced with a plethora of incongruent interfaces. For example, in order to write a program, a user may have to learn three languages: an operating system command language, an editor and a programming language. The error and diagnostic messages produced are often inconsistent even among the tools of a single manufacturer. Although the tasks of various tools diverge sufficiently so that some variation in interfaces is necessary, more uniformity could be achieved, especially for a set of tools that is to be used in a particular environment.

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

- The <u>dissonance</u> in the information (software-related project information), that is used and produced by various tools, allows valuable data to be irretrievably lost rather than communicated between the tools. The loss results because tools do not share a standard organized method of storing

Unix is a registered trademark of Bell Laboratories.

and communicating information to other tools that could profitably use use such data. For example, even though a typical compiler produces a parse tree data structure from the source file, a "pretty printer" tool to format the output probably will reconstruct the structure of the program from the source file. This reconstruction leads to 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 associated programming effort is considerable. (See figure 2.) Consequently, tools that should communicate via shared information, often never do.

- The level of granularity, that is, the size of the pieces, 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 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 standard text-editor tool uses information at the character level of granularity, but when those characters represent a computer program, the editor has no knowledge of the syntactic structure of that information. The individual or tool that produced the information was aware of the syntactic structure but had no standard method of recording the structure and communicating it to a subsequent tool requiring the information. Editing of characters

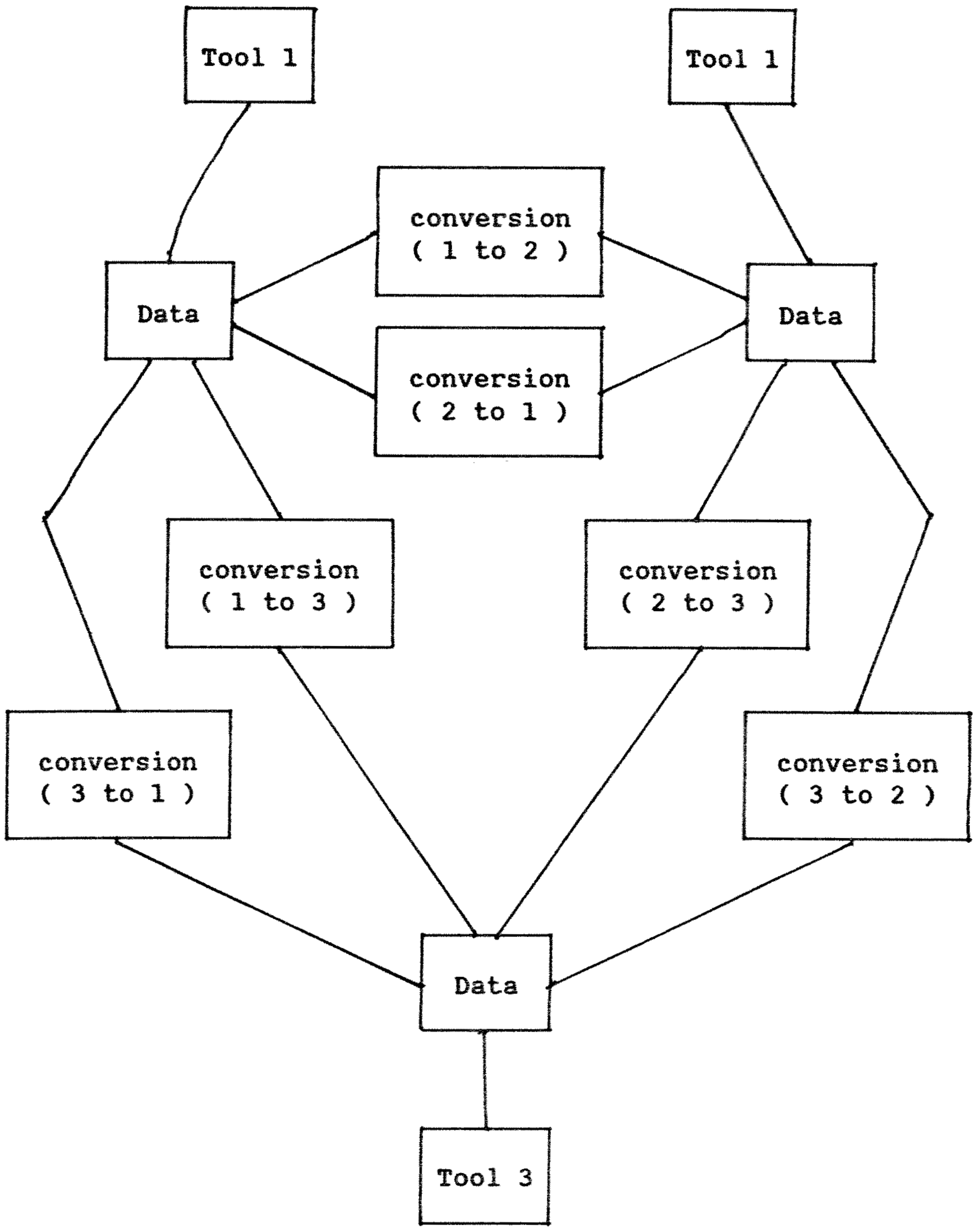

Figure 2: Conversions between Each Pair of Tools in an Un-Integrated System

to correct one error may inadvertently introduce another error. Syntax-directed editors have solved this problem in one domain by giving the editor a knowledge of the structure of the language [1,8,15].

Therefore, a set of software development tools is made more effective and potent by cooperating and communicating. (See figure 3.) 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 [2,13]. A tool which operates incrementally is able to use results from previous applications of the tool to avoid replicating work. 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 mentioned previously, the use of a common format for the information. Such a standardization may require either a change to a tool's design or 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 set of tools should have a common user interface in order to facilitate effective use. This again imposes the need either to redesign 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 tool sets. In addition to these negative motivations for the integration of tools, some compelling advantages of a more positive nature can be recounted.

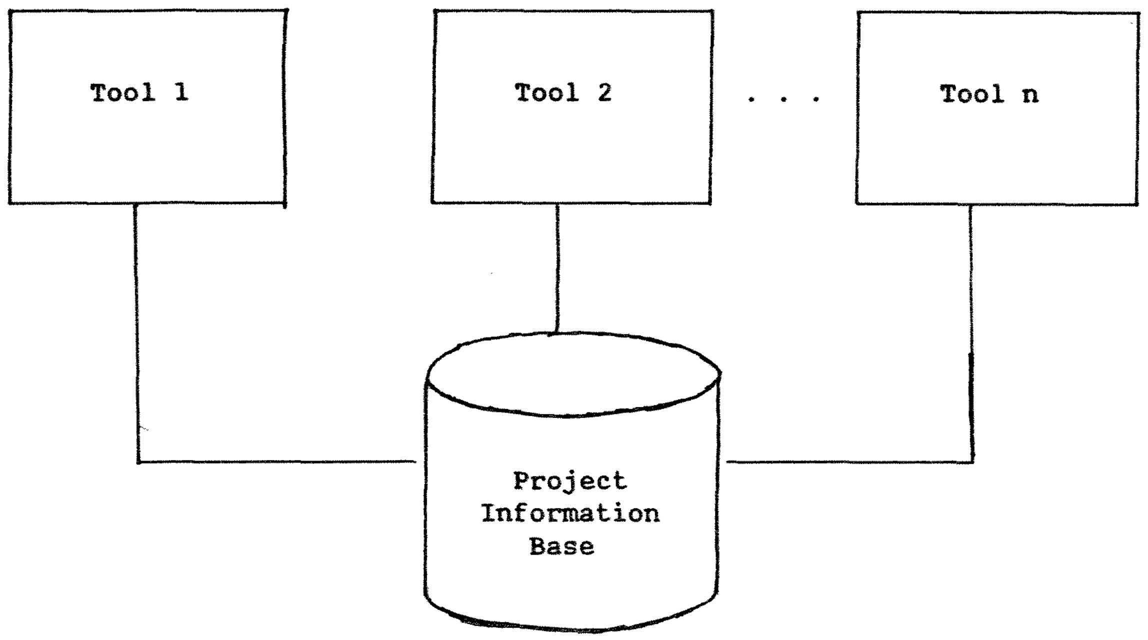

Figure 3: Communication and Cooperation through Tool Integration

2.2 Benefits of Tool Integration

The conceptual benefits of tool integration are dual. First, cooperating tools have a synergistic effect which is often 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, commonly used sequences of commands to multiple tools can be combined to form a more powerful command.

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, common diagnostic and help messages 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 concentrated attention on the project information. Ultimately, the user no longer is aware of the tools, but can focus on the project information.

3. The Parameters for, and a Categorization of, the Degree of Tool Integration

3.1. 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
- software development tools
- 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 [5]. (Object-oriented architectures tend to distribute the information base across many objects in the form of private stores, although an information repository object is possible [4].)

3.2. 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 the following three parameters:

- granularity
- cohesion
- harmony

The granularity of an object (e.g., a tool or an information base) 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 (examples follow in the next section).

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 maintains 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 they use, they are dissonant regarding that cohesion. A high level of tool integration is easier to attain among components which are harmonious.

3.3. 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 applied to each of the three system components--monitor, information base, and tools. Some of these cases, i.e. a specific parameter applied to a particular system component, will be more apropos to the categorization of tool integration than others. All will be mentioned with special emphasis on those that especially pertain to categorizing tool integration.

3.3.1. 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. An example is the contrast between the granularity of information in a database and the information in a file system. The database information generally has a finer granularity since the groupings of information to which the database has access are records which are relatively small chunks of information that may be spread across several files. Conversely, the file system has access to larger groupings of information in the form of files. The file server has a more coarse granularity of information. A coarse granularity for the information base does not imply that a user or a tool cannot find the logical piece of information needed, but that the search for it will be broader and have less automated help.

Tool granularity has several valid interpretations. The first interpretation relates to 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, namely 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 granularity is that of the granularity of tool interaction with the user. Granularity could be considered a continuum from batch to interactive. A batch-oriented tool has the coarsest granularity possible, i.e. no user interaction after the initial command. A highly interactive tool like a screen editor is typical of the other end of the continuum. 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 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 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 duplicating work begun on a previous invocation. An incremental tool most often consumes and produces information with a fine granularity. However, the crucial factor is the granularity of <u>control</u> of the tool. The control algorithm for an incremental tool is organized to operate in small increments rather than requiring complete execution 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 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. The advantages of a fine granularity of control are demonstrated by the incremental compilers [2,13] as compared to either traditional compilers, or to interpreters. Incremental compilers have flexibility in development of interpreters while generating the object code of a compiler.

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. However, 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 tool by automatically providing some input to the 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 and not of each tool. A coarse granularity of the interface, in which the monitor supplies a portion of the input, results in a more human-engineered environment. It is 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.

3.3.2. 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 impose 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). Tools are more easily integrated into an environment whose information base has both fine granularity and high cohesion. As a result, tools have access to a greater aggregation of more useful information.

Cohesion of a tool has a dual meaning. The first meaning is a concept parallel to that of information base cohesion: 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 than a tool with a coarse information granularity. The most important feature of the "pipe" mechanism of Unix [6] 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 increased level could be classified as a "high-level pipe" through which information is exchanged in two forms, project information and relational information.

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 duplicating 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 store 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 [16] mechanism that not only searches a list of known commands to find the closest match for a user command, but also examines the user's dynamic context. (DWIM is an acronym for the "Do What I Mean" mechanism of the Interlisp Programming Environment.)This examination would include such items as the last tool used, the parameters passed to this tool last, etc.

3.3.3. 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 has the greatest affect on tool integration and will be described in detail. A more cursory examination of the remaining two pairings will be presented for completeness. The amount of agreement between the tools and the information base also explains the amount of interaction between tools since that tool-to-tool interaction generally occurs through the information base.

Tool-information base interaction has two components, a syntactic and a semantic one. Syntactic harmony refers to agreement in the information representation of these two components. In particular, this agreement should occur in both granularity and cohesion. The information representations of two tools are said to be syntactically harmonious if each is harmonious with the information base. 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.

Syntactic dissonance must be overcome in order to integrate a tool into a set of tools. A mismatch in either granularity or cohesion of information results in the loss of critical information.

Semantic harmony describes the agreement in meaning between the information used by the tool and that used in the information base. Semantic dissonance is more problematic in that this type of dissonance must be overcome to integrate tools and yet general techniques for solving this problem are not easily specified. Semantic harmony or dissonance can be observed in at least three applications. In order of increasing importance (i.e. increasing difficulty of surmounting), these applications are as follows:

1. semantics of the information representation
2. semantics of the level of abstraction
3. semantics of the project information itself

The same project information can be represented in a descriptive or transformational manner. The semantics of the information is identical, however the meaning of the <u>representation</u> is quite divergent. Management information most often is recorded descriptively in details delineating estimated time to completion, resources budgeted, etc. Conversely, some version control systems [17] 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 in the recording of a description of all versions. Although conversion from one form to the other is more than a change in syntax, the same meaning can be represented in either method. (This dichotomy has been discussed by others in terms of procedural and descriptive methods of information representation [9].)

The harmony or dissonance of abstraction of information produced or used by a tool, and information stored in the information base, affects the integration of that tool. This level of abstraction can vary from 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 the information base, correction may be quite difficult. Although it is possible to convert from a low level of abstraction, in which much detail is present, to a high level of abstraction, conversion in the opposite direction is virtually impossible.

The most difficult type of dissonance to overcome is the actual semantics of the project information base. If a common ground does not exist between two tools or between a tool and the information in the repository, significant integration cannot be achieved, i.e. cooperation between the tools. Often the problem is not a complete lack of commonality, but rather, that the junction of the semantics 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 descriptions seem entirely incongruous only if one is unaware of their commonality. As our understanding of a particular field deepens, 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.

Overcoming semantic dissonance of any of the three types mentioned above is quite 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 the 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 [3,4], 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 [5]. For example, the information base of the TRIAD software environment is structured as a tree of forms [7]. This tree models the underlying development method which reflects the general order in which this information is presented to, and requested from, the user.

3.4. Categorization of Integration – Loose versus Tight

The confluence of the 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. The terms "tight" and "loose" integration actually refer to the extremes of the continuum. 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. A system is categorized as tightly integrated if it utilizes the following features: 1) fine granularity and high cohesion of the information base and the tools, and 2) syntactic and semantic harmony of the tools to achieve a high degree of inter-tool communication and cooperation. Poe [1], Pecan [12] and the Cornell Program Synthesizer [15] are examples of tightly integrated systems.

The advantages resulting from a tightly integrated system are numerous. First, tools are more efficient since there is less duplication of effort. The tool does not have to reparse input to recover the structural information since tools store their results in, and take their input from, the information base.

Secondly, the tool's response time to the user tends to be short for incremental tools since the tool is taking a small step each time it is invoked. This enables the user's attention to be focused on the project information and the task at hand rather than being distracted by long waits for service. As pointed out in the Magpie system [13], the power of today's computer systems is sufficient enough that servicing users often leaves computer time (i.e. CPU cycles) available between user keystrokes and user thinking. This time can be used effectively to incrementally apply tools; as a consequence, the user has access to more up-to-date data.

Not only are tool results available more quickly, particularly in the case of incremental tools, they are also 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 also is 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 tight degree of integration (i.e. the user cannot adjust the increment size for incremental tools) and the volume of information available from frequent interactions with the tool 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. Loose integration is a degree of cooperation between tools in which the granularity is more coarse and/or the cohesion is lower, and 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 [10,11] 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) and the information produced by the tools is quite coarse. The tools produce information in a similar coarse granularity. The amount of cohesion present in the tools and the information base is minimal. Yet, Toolpack is able to provide a certain level of tool cooperation which eases the user's task.

For tools whose purpose requires little user interaction, e.g. many batch-oriented tools, loose integration is most appropriate. For any set of 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 amount 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 non-rigorous, exploratory project development.

These examples have been at the extremes of the tight-loose continuum. There is a whole range of intermediate levels with various combinations of parameter values.

3.5 A Method for Evaluating Integration Potential

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. However, a precise, quantitative measure of the integration potential of a tool is beyond the current state-of-the-art. Figures 5, 6, and 7 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 some form of information repository and a user interface or monitor.) A tool which has negative answers to one entire set of questions (e.g. to all the tool granularity and cohesion questions) and affirmative for the remaining questions has less potential than a tool whose negative evaluations are dispursed throughout all the questions.

INFORMATION REPOSITORY

<u>Granularity</u>

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?

<u>Cohesion</u>

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

<u>Granularity</u>

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?

<u>Cohesion</u>

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 5: **An Informal Method for Evaluating a Tool for its Integration Potential – Granularity and Cohesion of the Information Repository and the Tools**

MONITOR

<u>Granularity</u>

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

<u>Cohesion</u>

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

Figure 6: **An Informal Method for Evaluating a Tool for its Integration Potential – Granularity and Cohesion of the Monitor**

TOOL – INFORMATION BASE

<u>Syntactic Harmony</u>

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?

<u>Semantic Harmony</u>

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 7: **An Informal Method for Evaluating a Tool for its Integration Potential – Harmony versus Dissonance**

4. Conclusion

The integration of existing software tools is a technique which can multiply the functionality of a set of tools (i.e. the synergistic effect) and can increase their usability by permitting the user to maintain a focus of attention on the problem information rather than the tools. The level of integration has been parameterized by the terms granularity, cohesion, and harmony. These parameters have various meanings when applied to specific components of the software development environment. An informal method has been given to characterize the integration potential of a given tool.

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, and even the user's mental state 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.

REFERENCES

[1] C. N. Fischer, G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock, "The Poe Language-Based Editor Project," Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, pp. 21-29, April 1984.

[2] P. Fritzson, "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, Pennsylvania, pp. 113-123, April 1984.

[3] A. Goldberg, Smalltalk-80 The Interactive Programming Environment, Addison-Wesley Publishing Company, 1984.

[4] A. Goldberg, "The Influence of an Object-oriented Language on the Programming Environment," Proceedings of the ACM Computer Science Conference, ACM, pp. 35-54, February 1983.

[5] S. M. Kaplan, M. T. Harandi, S. N. Kamia, R. H. Cambell, R. E. Johnson, J. Liu, and J. Purtilo, "An Architecture for Tool Integration," Proceedings of the Workshop on Advanced Programming Environments, Trondhiem, June 1986.

[6] B. Kernighan and R. Pike, The Unix Programming Environment, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

[7] J. C. Kuo, C. Li, and J. Ramanathan, "A Form-Based Approach to Human Engineering Methodologies," Proceedings of the 6th International Conference on Software Engineering, IEEE Computer Society, Toyko, Japan, pp. 254-263, September 1982.

[8] R. Medina-Mora and D. S. Notkin, ALOE Users' and Implementors' Guide, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1981.

[9] M. L. Minsky, "A Framework for Representing Knowledge," In Patrick Henry Winston, Ed., The Psychology of Computer Vision, McGraw Hill Book Co., 1975.

[10] L. J. Osterweil and W. R. Cowell, "The Toolpack/IST Programming Environment," Proceedings of a Conference on Software Development Tools, Techniques, and Alternatives, IEEE Computer Society, Washington, D. C., pp. 326-333, July 1983.

[11] L. J. Osterweil, "Toolpack - An Experimental Software Development Research Project," IEEE Transactions on Software Engineering 9, 6 pp. 673-685, November 1983.

[12] S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, Pittsburgh, Pennsylvania, pp. 30-41, April 1984.

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

[14] M. Shaw, "Beyond Programming-in-the-Large: The Next Challenges for Software Engineering," Technical Memorandum SEI-86-TM-6, Software Engineering Institute, Carnegie-Mellon University, May 1986.

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

[16] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," Computer 14, 4, pp. 25-33, April 1981.

[17] R. M. Thall, "Large-Scale Development with the Ada Language System," Proceedings of the Computer Science Conference, Association of Computing Machinery, Orlando, Florida, pp. 55-67, February 1983.

Figure 1: **Un–Integrated Tools Lack Cooperation**

Figure 2: **Conversions between Each Pair of Tools in an Un–Integrated System**

Figure 3: **Communication and Cooperation through Tool Integration**

Figure 5: **An Informal Method for Evaluating a Tool for its Integration Potential – Granularity and Cohesion of the Information Repository and the Tools**

Figure 6: **An Informal Method for Evaluating a Tool for its Integration Potential – Granularity and Cohesion of the Monitor**

Figure 7: **An Informal Method for Evaluating a Tool for its Integration Potential – Harmony versus Dissonance**

FOOTNOTES

J. D. Kiper is with the Department of Systems Analysis, Miami University, Oxford, Ohio 45056.

Unix is a registered trademark of Bell Laboratories.

For correspondence, contact the author at the following address:

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

(513)529-5938

The Integration of Software Development Tools

James D. Kiper, Ph.D.

April 1987

Systems Analysis Deparmtent
Miami University
Oxford, Ohio 45056