Software Tools and Environments

STEVEN P. REISS

Brown University, Providence, Rhode Island ⟨spr@cs.brown.edu⟩

Any system that assists the programmer with some aspect of programming can be considered a programming tool. Similarly, a system that assists in some phase of the software development process can be considered a software tool. A programming environment is a suite of programming tools designed to simplify programming and thereby enhance programmer productivity. A software engineering environment extends this to software tools and the whole software development process.

Software tools are categorized by the phase of software development and the particular problems that they address. Software environments are characterized by the type and kinds of tools they contain and thus the aspects of software development they address. Additionally, software environments are distinguished by how the tools they include are related, that is, the type and degree of integration among the tools, and by the size and nature of the systems they are designed to address.

Software tools and environments are designed to enhance productivity. Many tools do this directly by automating or simplifying some task. Others do it indirectly, either by facilitating more powerful programming languages, architectures, or systems, or by making the software development task more enjoyable. Still others attempt to enhance productivity by providing the user with information that might be needed for the task at hand.

SOFTWARE TOOLS

A wide variety of software tools have been developed. Programming tools are typically considered to be those used explicitly during the actual coding process. This involves both the creation of new code and the edit-compile-debug cycle that is typically used in debugging code. These include:

Program Editors. These range from simple text editors to syntax-directed or language-sensitive editors that know the syntax of the programming language being used and attempt to minimize the amount of typing necessary and to format the resultant text. Perhaps the most common type of program editor is a compromise between these two, a language-knowledgeable editor. Here a text editor is augmented with language knowledge to provide such capabilities as automatic indentation, parenthesis checking, and even simple cross-referencing. The primary example of such an editor is *emacs* [Gosling 1982].

Compilers. These convert source code into a machine-executable form, either actual machine code or an internal form that can be interpreted. Compilers differ in how much optimization they perform, and whether they are incremental (i.e., compile only what has changed) or batch.

Linkers and Loaders. These combine compiled files with libraries to produce an executable file. Because the size of binaries has been increasing, link time now comprises a significant fraction of the edit-compile-debug cycle. Incremental linkers, which modify only those parts of the executable that have changed, attempt to alleviate this.

Preprocessors. These provide capabilities beyond the basic source language. Some, such as the C preprocessor, are built into the source language, while others such as lex [Lesk 1975] or yacc [Johnson 1974] provide languages of their own.

Cross Referencers. These tools provide a means for relating the use of a name to its definition. Cross-reference information is most efficiently produced by the compiler, although some systems do fast approximate scanning to generate the necessary information. Cross-reference information can be used in an editor (tag information in *emacs*) or in an independent system with either a textual or a graphical display. Common graphical cross-reference tools provide displays of the call graph and the class hierarchy graph for object-oriented programs.

Source-Level Debuggers. These allow the user to control the execution of the program, setting breakpoints and exploring values as needed. More sophisticated debugging tools can handle optimized code and libraries as well as multiple languages in the same system. These have been augmented with visual displays of program values, allowing the programmer to see the application's data structures in their conceptual form.

Debugging Aids. These are tools that provide either compile- or run-time checking beyond that offered by the programming language. These include systems and libraries for detecting problems in the use of heap memory, such as Purify [Hastings and Joyce 1992], and systems as *lint* [Ritchie et al. 1978] that go beyond compiler checks to find legal constructs that might be potential problems.

A second set of tools addresses software engineering issues. These tools are concerned with maintaining the overall system rather than coding itself. Existing tools in this area include:

System Builders. These allow the user to define a model of the system that shows how the system should be built. The model includes information on dependencies, compilation options, and what commands should be executed to build each binary. The tools are then able to build the original system or to update a system incrementally based on a set of source file changes.

Version Managers. These tools allow multiple versions of a source file to exist simultaneously. This permits parallel system development to allow multiple programmers to cooperate, to allow older released versions of a system to be maintained while a new version is developed, or to allow customized versions of a single system to coexist.

Design Editors. These let a user design a system using a variety of graphical design notations. Commonly known as CASE (Computer-Aided Software Engineering) tools, these have been developed for such representations as Petri nets, SADT [Ross 1985], statecharts [Harel 1984], and object-oriented design using OMT [Rumbaugh et al. 1991]. Many of these tools generate at least a code framework based on the design. If enough information is provided with the design, some of these tools can simulate aspects of the system, allowing the developer to test the design at a high level.

Code Generators. These, also known as fourth-generation languages (4GLs), are actually special-purpose high-level languages that let the programmer interactively specify a large portion of a system without having to code it. They are most commonly used for defining user interfaces and the interaction of a program with a database system.

Testing Aids. These are tools that attempt to automate the process of testing software systems. They range from test case generators that analyze source code or specifications to generate a suite of test cases, to regression testing systems where the program-

mer generates the test cases but the system does the bookkeeping involved with running each test case, determining if it succeeded or failed, and reporting the result.

INTEGRATING TOOLS INTO AN ENVIRONMENT

Software tools can be combined in a variety of ways using various integration techniques. Early environments were either loose confederations of tools such as UNIX or single systems that combined all the relevant tools. Single-system environments have the advantage of allowing the tools to be tightly coupled so that the programmer is aware of the environment and not the separate tools. They have the disadvantage that they are typically closed systems in which it is hard to incorporate new tools or to use multiple languages or existing code, and are relatively large systems. Federated environments solve many of these problems—they are typically open systems in which it is easy to develop and utilize new tools and in which systems can be built using multiple languages and libraries. However, they put the onus of using the tools in the right order and the right way on the programmer and do not provide a common framework.

Three approaches have been used to integrate single-system environments with the openness and flexibility of federated environments. These involve ways for the tools to share information and interfaces.

Data integration assumes that the federated tools share information. This typically involves the development of a database or repository to hold the information that needs to be shared among the tools. This has the potential to allow a high degree of integration and to simplify the various tools by having them share the work. Typically, the compiler in such an environment generates an intermediate representation (generally abstract syntax trees) that is stored in the database and can be used directly by other tools that would otherwise have to parse the source. The disadvantage of this approach is that the database required is quite large and a sophisticated database system is necessary to make the environment work. Also, such environments tend to be closed rather than open, both because of the need to integrate any new tool with the database system and because the database itself tends to be designed for a particular language.

A second approach to integration involves the use of a common front end. The most prevalent tool in software development is the text editor, used for creating programs, documentation, system models, and so on. It is relatively easy to embed in an editor commands that invoke other tools and use the output of those tools. Thus *emacs* allows the user to invoke the system-build package *make* [Feldman 1979], puts the output of the resultant compilation in a buffer, and allows the user to use that buffer to go to the source lines where errors occurred. It also lets the user invoke the debugger and other tools. This approach gives some sense of integration by providing a common interface for a variety of tools. Moreover, it allows new tools to be integrated quite simply, provided that they have a textual interface. The approach does not provide all the benefits of integration, however, in that tools really do not share information and the programmer is still aware of the different tools and their uses.

The third approach to integration, control integration, involves message passing between the tools. Here tools send messages to other tools whenever they need to share information or whenever a command from one tool is invoked from another. Rather than point-to-point messaging, this is generally organized using a central message server as an intermediary. Each of the tools tells the message server what messages it is interested in. Tools then send messages to the server, which then sends them selectively to all recipients that had expressed interest. This approach provides a high degree of imme-

diate integration. It allows multiple tools to operate as if they were one, for example facilitating the use of an editor to invoke other tools or allowing all current tools to display a common focus in response to a user action. What it lacks is the ability of tools to share information over time and the ability of tools to share large amounts of information such as the syntax trees for a system.

Actual environments combine these different integration approaches in various ways. The PCTE standard [Boudier et al. 1989], for example, uses control integration and a common set of front-end utilities on top of a data integration basis. FIELD [Reiss 1994] and related environments primarily use control integration but provide specialized repositories to hold long-term data such as cross-reference or configuration-management information.

FUTURE TOOLS AND ENVIRONMENTS

Because software development is a difficult and time-consuming process, people continue to develop and extend software tools and environments. What we can expect in the future is better versions of the existing tools: faster compilers, incremental loaders, better and more diverse debugging aids, improved editors, and so on. We also expect to see better environments that combine a high degree of tool integration with openness to new tools and languages. In addition, current research is aimed at extending software tool support to new domains that include:

Process Tools. These are tools that assist in managing the software development process. They allow the process to be defined using a combination of rules and procedures and then provide assistance in bookkeeping and managing the process.

Groupware Tools. These are tools that allow multiple programmers to work together in a controlled way. While version management provides a basis for multiple programmers, the new tools will facilitate enhanced levels of cooperation and interaction among developers at distributed sites.

Visualization Tools. The current visualization efforts, notably class hierarchy displays, call graph or module displays, and limited data-structure visualization, are being enhanced to provide high-quality displays of large amounts of program data in a meaningful way. These will become a standard part of software environments.

Program Analysis Tools. These are tools that undertake a detailed semantic analysis of a system and provide user feedback. The feedback can relate to potential problems, mismatches between code and specifications, assistance in merging different versions, or assistance in reengineering code.

REFERENCES

BOUDIER, G., GALLO, F., MINOT, R., AND THOMAS, J. 1989. An overview of PCTE and PCTE+. *SIGPLAN Not. 24,* 2, (Feb.), 248–257.

FELDMAN, S. I. 1979. MAKE: A program for maintaining computer programs. *Softw. Pract. Exper. 9,* 4, 255–265.

GOSLING, J. 1982. *Unix Emacs.* Carnegie Mellon Computer Science Dept. Pittsburgh, PA (Aug.).

HAREL, D. 1984. Statecharts: A visual approach to complex systems. Dept. of Applied Mathematics, Weizmann Institute of Science.

HASTINGS, R. AND JOYCE, B. 1992. Purify: Fast detection of memory leaks and access errors. In Proceedings of *Winter Usenix Conference* (Jan.).

JOHNSON, S. C. 1974. YACC—yet another compiler compiler. CSTR 32, Bell Labs., Murray Hill, NJ.

LESK, M. E. 1975. LEX—a lexical analyzer generator. CSTR 39, Bell Labs., Murray Hill, NJ.

REISS, S. P. 1994. *FIELD: A Friendly Integrated Environment for Learning and Development.* Kluwer, Norwell, MA.

RITCHIE, D. M., JOHNSON, S. C., LESK, M. E., AND KERNIGHAN, B. W. 1978. The C programming language. *Bell Syst. Tech. J. 57,* 6, 1991–2020.

ROSS, D. T. 1985. Applications and extensions of SADT. *IEEE Computer 18,* 4 (April), 25–35.

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., AND LORENSEN, W. 1991. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ.