Chaired by: R.E. Fairley
Panelists: G.P. Brown, E.S. Cohen, M.A.F. Muellerburg, M.L. Powell, and H.L. Wertz

The session on debugging in integrated environments started with Fairley's characterization of integrated environments as follows: (1) versatile collection of analysis, design, implementation, testing, and maintenance tools; (2) consistent user interface among the tools; (3) common representation of information (e.g., Stoneman's database model, Unix pipe); (4) history/version control/configuration management capability; and (5) encouragement of good practice. He then asked the audience the following questions: (1) what is your view of environments? (2) what possibilities exist for debugging in integrated environments that do not exist in a mere collection of tools? (3) what is the difference between high-level debugging and low-level debugging with high-level tools?

## The Role of Debugging within Software Engineering Environments

by M.A.F. Muellerburg, GMD, Software-Technologie

Two kinds of software development environments can be distinguished: (1) an individual programmer developing small to medium scale programs. (2) a team of programmers constructing large-scale systems. Programming environments (PEs) support the first kind of software development and provide tools and methods for a particular programming language. Software engineering environments (SEEs) supports the second kind of software development. SEEs offer tools and methods for the technical tasks of software development, such as requirements analysis, system specification and validation, and also for the further tasks of project management and software preparation. They provide tools, models of the product and of the production process, as well as means for representation. The usefulness of such systems depends on the particular tasks and constraints of a software development project.

There are two main problems for debugging: (1) understanding the software (i.e., its structure and behavior) and (2) estimating the impacts of an intended change. Debugging in the context of PEs and SEEs is based on a programming language.

Many PEs consist of an editor, compiler or interpreter, linker, loader, run-time system, and debugger. Each of the tools knows the language and its constructs. So syntactical errors can be prevented on entry and the user can interact with the system on the level of language concepts. A significant change in the programmer's situation, and thereby also in debugging, are already seen today. For example, software development within PEs is highly interactive; high-resolution display terminals provide windowing and mouse techniques; and powerful small computers allow work stations to be tailored to particular tasks.

SEEs may evolve into expert systems to provide information about the software and development tasks (e.g., test cases for previously detected errors). Furthermore, a query facility may be used instead of reading large source listings; e.g., to display where a function is used.

Future software development will be influenced by (1) terminals allowing window and mouse capability; (2) small efficient personal work-stations; and (3) networks of personal work-stations. Expert systems will (1) help detect errors; (2) suggest possible causes of errors; and (3) propose corrections. The most significant change may come when knowledge engineering and functional programming replace present software engineering techniques based on procedural programming. Here, specifications will be executed. Furthermore,

validation and debugging will focus on specifications rather than on programs. So dynamic analysis and debugging will only check the remaining problems with respect to program behavior on a machine; e.g., overflows.

(Audience) Isn't the distinction between PEs and SEEs an artifact?
(Speaker) No, there are indeed fundamental differences between environments for a single programmer and for a group of programmers.

## An Integrated, Interactive, and Incremental Programming Environment

H.L. Wertz, Universit Paris

Wertz described an integrated, interactive, and incremental LISP programming environment. The reasons for integrated environments are (1) for developing programs; (2) for executing programs; and (3) for interactively modifying programs. In his paradigm, a program is a set of different versions, documentation, and possible active annotations. The system supports the following. (1) The programmer can attach input/output assertions to every point in the program. (2) The system incrementally constructs static and dynamic documentation of the program. This documentation is usable by the debugger. (3) There are three modes of evaluation: normal, symbolic, and careful evaluation, where careful implies a checking of all the attached assertions and/or stepping and tracing at variable grain size. (4) While modifying the program, the system does some measuring; e.g., the scope of modification. (5) There is complete version control for editing and execution; e.g., the programmer can execute or develop previous versions without influencing the current one.

The major advantage of the system is that all annotations to a program are hidden to the user, only the effect is visible; that is, the program itself never appears to be modified.

(Audience) Why don't systems such as this get used much?
(Speaker) Programmers do not know of the systems.
(Audience) Why don't people build system like this for Pascal?
(Speaker) It is easier to maintain LISP programs. People who build and use such systems are used to programming in LISP, not Pascal.
(Audience) LISP has simple syntax and semantics. LISP environments are relatively easy to construct and modify.

## Program Visualization

by G.P. Brown, Computer Corp. of America.

Brown's thesis in designing and implementing the program visualization (PV) environment is that: (1) debugging requires good information about the behavior of a system; (2) diagrams are an effective (often the most effective) way to present information; and (3) a software development environment should support the production and use of graphics. She then showed slides (1) for spatial navigational aids to present integrated views of the static information about a program and (2) for animated views of a running program.

One of the motivation for the PV project was the experience of another project at CCA that built SDD-1, a distributed database system. The SDD-1 project team built a high-level integrated graphic display of transaction processing in SDD-1. This dynamic graphic view of the system was originally intended as a demonstration tool, but it ended up being used in-house as a debugging tool. The SDD-1 display was, however, custom tailored, and in general, dynamic graphics displays are hard to build and maintain. The PV project is addressing this problem by constructing a general graphics production environment to support the construction of graphic debugging tools.

(Audience) How useful is color and how expensive is it?
(Speaker) Very useful but very expensive.
(Audience) In the text of programs, color does not seem to buy much.
(Audience) Using color to show time-steps seems most effective.
(Audience) Use color to show the frequency of code execution; e.g., using warm and cool colors.
(Audience) What makes prodduction of graphics hard?
(Speaker) Construction of high-level integrated views of system structure and behavior and design of graphic representations are hard problems.

## A Database Model of Debugging

by M. L. Powell, University of California, Berkeley

An integrated environment improves command syntax, world model, and debugging environment. In OMEGA, all program information is stored in a relational database system. The database understands the construction of a program (e.g., source, linkage, heritage) and the execution of a program (e.g., run-time state, history). The system includes debugging capabilities to

provide the programmer with a simple yet powerful mechanism for describing requests. Debugging in OMEGA is a sequence of queries and updates on the database.

(Audience) A relational database is nice since a new relation can be dynamically added. Is a relational database essential?
(Speaker) No.
(Audience) How much information needs to be stored? In particular, some information is hard to capture.
(Speaker) The point is that if some information is hard to gather, we probably do not need it.

An Extensible Debugger
by E.S. Cohen, Brandeis University

It is important for a debugger to provide a well-defined set of primitive functions which, using an extension language, can be extended with user-defined functions. The debugger must provide access to a database holding both static and run-time information. Furthermore, it must provide the ability to define hooks which specify an event and an action to be executed when that event occurs, as well as the ability to selectively enable and disable events. It is possible to define functions for, among other things, tracing, single-step execution, generation of execution histories, and program animation. As an example, he showed how a hook can be defined and enabled in his extensible debugging language based on LISP.

(Audience) Programmers want to spend less time changing their programs. Do you expect them to change a debugger?
(Speaker) A library of predefined debugging routines will be provided for such programmers.
(Audience) Isn't your debugging language another language?
(Speaker) Yes, but it is a uniform language that can be used for multiple languages.

Insup Lee
Computer Sciences Department
University of Wisconsin - Madison
Madison, Wi 53706