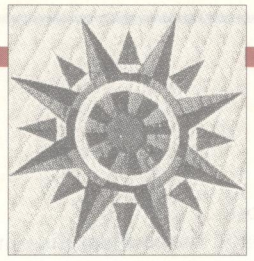


The Symbolics Genera Programming Environment

Janet H. Walker, David A. Moon, Daniel L. Weinreb**, and **Mike McMahon
Symbolics, Inc.

This Lisp-based system helps designers get from prototype to product faster. The key is an open architecture and highly integrated development tools.

Symbolics Genera is a mature software development environment that integrates features normally found in an operating system, its utilities, and the applications running under it. An environment intended for developing large, complex software systems, Genera also supports its own development and maintenance as a commercial product.

Genera's primary development language is Lisp, which is also its implementation language. The environment we describe in this article supports development in other languages, including Pascal, C, Fortran, Ada, and Prolog, but we concentrate on Lisp.

Genera runs on hardware designed to run compiled Lisp code and other languages efficiently.[1] The hardware environment has features that are critical to the software environment:

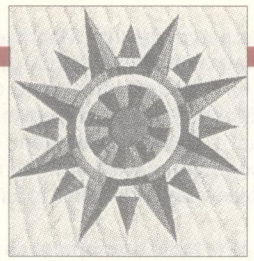
- A single-user workstation with a high-resolution, bitmapped display and a mouse. The display includes text, graphics, and presentations of objects.
- A high-speed, local-area network for communication among workstations and access to servers.
- Hardware checking of data types, array subscript bounds, and uninitialized variables, performed in parallel with computation and memory access.
- Hardware-assisted automatic storage allocation and reclamation.

Genera evolved expressly to provide full life-cycle support for major development projects. It includes facilities for exploration, development, communication, consolidation, documentation, testing and refinement, distribution, maintenance, and retirement. A detailed description of all these facilities is beyond the scope of this article; our emphasis is on development.

Development methodology

The development methodology supported by Genera could be described as *rapid productizing*, the rapid evolution of software from early working versions to final product.

0740-7459/87/1100/0036/$01.00 ©1987 IEEE



Some conventional views of software development separate design, prototyping, and coding as independent, serial stages.[2] While this division is useful for conceptualizing the variety of activities in a software project, we believe it falls short of describing the actual software-development process.

In our experience, successful software products evolve throughout development: Early implementations lead to better understanding of requirements, refinement of design, more implementation, more refinement, and so on, until the requirements are both well-understood and well-implemented.

Evolutionary refinement. We follow a software development methodology of evolutionary refinement that has these major elements:

- Incremental design and implementation. We focus on the essential requirements first and elaborate on the details later, when the problem becomes better understood as a result of the rigor enforced by implementation.
- Parallel refinement of requirements, design, and implementation. The end product reflects a good understanding of the requirements, rather than features piled onto an early design that was based on an incomplete, initial understanding of the requirements.
- A single approach. We use the same language for the prototype and the product. There is no point at which the prototype is "done" or discarded; there is no loss of work or momentum from having to change methods or languages during the project.
- A functional prototype. Many people regard a prototype as a facade — something that looks like the final product but either does not work at all or does not work like the final product. We view a prototype as a perspective on the progress of the project rather than a separate effort. Code produced early in the design cycle *works* and often survives in the final product. This functionality is important, to understand the project requirements and further refine the design.

Complex problems. The evolutionary methodology is particularly well-suited to complex problems because it is in these situations that requirements are difficult to state and interactions difficult to understand.

We see two kinds of complex problems, both of which are particularly suited to development in our environment: (1) a new problem that no one has ever written a program to solve; and (2) a very large problem with pieces that are simple enough in themselves, but made complex by the sheer number of pieces and possible combinations. The second case is a matter of information-management complexity, such as when dealing with the separate components of a several-hundred-page document, not just numerical complexity, such as processing ever-larger image arrays.

Our major premise is that the software designer is a valuable and limited resource.

Requirements and principles

Our major premise is that the software designer is a valuable and limited resource. It is fundamentally important to the success of the project to use that person's time as effectively as possible. Furthermore, we assume that most or all the people involved in the project are designers (rather than coders) under the direction of an elite designer. The evolutionary methodology involves the designer in a tight design-evaluate-refine loop.

To date, our experience has been with project teams of about a dozen people. We are not talking about supporting design teams of 100 or 1000 people because we do not believe that larger groups can work effectively on the same large, complex project.

Our premise about the value of the designer's time leads to a further set of assumptions that guide the design of the development environment:

- Support for complexity. Complexity in a working environment is manifest at both the user and the artifact level. At the user level, programmers must engage in mental time-sharing; they do several tasks at once. At the artifact level, programmers need tools to manage the relationship of the many parts of complex programs.
- Support for community. People using independent workstations must be as close to each other as they would have been on a time-sharing system. They must transparently share files, electronic mail, entity names, incremental updates, and so on.
- Automate the tedious. Because it allows many sweeping changes to a developing system, the evolutionary methodology could be prohibitively expensive in human terms. To minimize the human cost, a system must provide facilities to manage the details of program structure and compilation.

Implementation principles

One level down from our requirements principles are the principles that guide our implementation choices:

- Data-level integration. All functions must be able to communicate through shared data structures.
- Support for incremental change. Constant, small changes must be supported in designs, in work procedures, and in quality enhancement.

Genera's roots

Genera is a descendant of software developed as part of the Lisp Machine project at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology (the AI Lab). The goal of the AI Lab is to support researchers doing innovative things with computers. When the Lisp Machine project began in 1975, the AI Lab's chief computational resource was a Digital Equipment Corp. KA-10-based time-sharing system. Its hardware constraints were limiting much of the work done at the AI Lab. The Lisp Machine project was undertaken to address the problem.[1]

The primary goal of the Lisp Machine project was to build a computer capable of high-speed Lisp execution in a large virtual memory, one that was both appropriate and practical to supply as a single-user computer to each researcher. The project focused on architectural support for efficient, compiled Lisp execution, while retaining the full error-checking of interpreted Lisp. The machines were designed to be connected by a local-area network for file access and resource sharing.

The software development environment was influenced initially by the AI Lab's system — which was based on the MacLisp language,[2] the Emacs text editor,[3] and the ITS time-sharing system — and had been augmented by many personal tools. Another influence was the early work on Altos done at Xerox PARC, from which we learned about windows, mice, and networks. We also borrowed ideas from other systems, including Multics and Tops-20.

The Lisp described here is Symbolics' implementation of Common Lisp,[4] extended with exception handling and Flavors.[5] Flavors provides the abstract data types and generic operations of an object-oriented programming language, like Smalltalk.[6] Many system facilities use the object-oriented programming paradigm, and tools in the environment support that paradigm as well as the conventional Lisp programming paradigm.

Symbolics was founded in 1980 to commercialize the Lisp Machine technology and to continue development of the hardware and software system. Since then, one major new architecture and four major software versions have been shipped. This article discusses concepts as instantiated in Genera 7.1, the most recent software release.

Genera is running on about 3500 systems. It represents more than 10 years of experience and refinement, based on intensive real-world use and user feedback.

References

1. R.D. Greenblatt et al., "The Lisp Machine", in *Interactive Programming Environments*, D.R. Barstow, H.E. Shrobe, and E. Sandewall, eds., McGraw-Hill, New York, pp. 326-352.
2. K.M. Pitman, "The Revised MacLisp Manual," Tech. Report MIT/LCS/TR-295, Massachusetts Inst. of Technology, Cambridge, Mass., May 1983.
3. R.M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," *SIGPlan Notices*, June 1981, pp. 147-156.
4. G.L. Steele, Jr., *Common Lisp: The Language*, Digital Press, Burlington, Mass., 1984.
5. D.A. Moon, "Object-Oriented Programming with Flavors," *SIGPlan Notices*, Nov. 1986, pp. 1-8.
6. J. Diederich and J. Milton, "Experimental Prototyping in Smalltalk," *IEEE Software*, May 1987, pp. 50-64.

• Open system. The software must never restrict the designer's grasp — all parts of the system must be available to designers for extension or replacement.

• Reusability. All parts of the system must be both available and locatable by designers to be used, modified, or extended by small modular additions.

• Extensibility. Users must be able to tailor the work environment to support their own styles and preferences.

• Self-revealing. Information about the internal workings of the system must always be visible or available.

Data-level integration

Genera can run any number of processes at once, all sharing the same virtual memory. This shared memory is treated as a set of objects, not as raw bits or bytes. The data used by programs and the program functions themselves are all objects in memory.

Simple objects include scalars (such as numbers and characters) and data structures (such as lists and arrays). More complex objects include application-specific types, which comprise two kinds of information: (1) state information, a collection of other objects, and (2) behavior information, a collection of functions that define the actions that can be performed on the objects.

Objects are self-describing in the sense that each object knows its data type. Examples of objects defined by the development environment include the source text for a section of a program, a compiled function, and a network of interdependencies among sections of a program.

Different programs can easily understand and use each other's objects because the memory they share contains these self-describing, application-oriented objects. In traditional systems, programs communicate through a pipe, an interchange format, or a clipboard. Figure 1a illustrates some models for sharing data between two programs. Figure 1b shows the shared-object model in more detail.

Multiple processes can work simultaneously on multiple activities in multiple windows and, because of data-level integration, all work on the same object base. As a result, traditionally separate utilities are very tightly integrated: The editor can invoke the mail-sending program on a buffer; the mail-reading program can invoke the compiler on a function definition in a message; the debugger can invoke the on-line documentation system to explain a function in the current frame, and so on.

Because these different tools operate on the same object in memory, the person using them does not have to perform any manipulations to make the data from one tool available to another. In this way, the software more nearly supports the human view of the program — a set of tangible objects that really exist inside the computer.

Having this level of data integration available encourages programmers to design generality into their data structures. As a result, their programs can integrate smoothly with other programs and can be extended easily in minor ways. This is the bedrock principle upon which the rest of the system is built.

Incremental change

Complex, experimental software systems are too large to work on all at once.

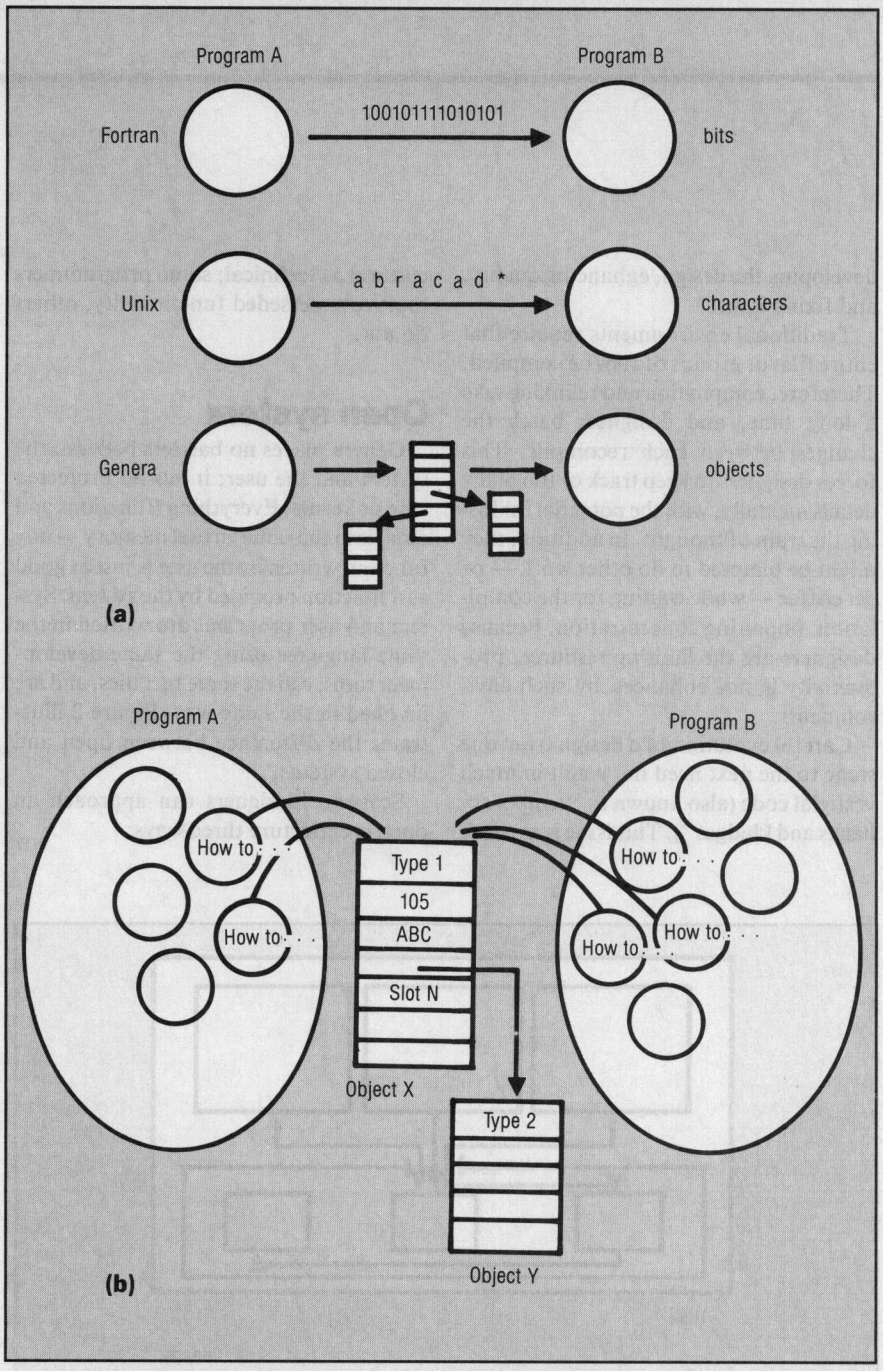


Figure 1. Comparing models of interprogram communication. **(a)** Programs can communicate in different ways; **(b)** in Genera, they use the same data for interprogram communication as they do for computation. This approach combines object-oriented shared memory with well-defined protocols for classifying objects and operating on them. Each object has a type (which it knows) and a series of slots that hold either simple data objects or other complex objects. The programs are made up of a set of functional modules. Different programs can contain different methods for operating on the same objects, represented with "How to . . ." labels.

Various strategies have been devised to help designers manage this complexity by dividing the work into small steps, each small enough so they can understand it fully. This divide-and-conquer principle emerges in our methodology in three areas: design, quality enhancement, and procedure.

Design. As the system design evolves, the designer transforms the software from one stable state to another, one step at a time. The software emerges as a sequence of programs, each of which can do something and is partially complete. At each step, the designer can run the program and see what the results of the changes were, using the feedback to decide whether to change direction or proceed to the next step. This cycle of incremental feedback and guided design is used not only for early experiments but for the entire program development.

It is important to realize that the designer must have an overall design for the system when this process starts. Incremental design does not mean that you make it up as you go along! It does mean that the overall design can be modified to a slightly different overall design, which might mean backing up and rewriting portions of the software.[3] Incremental development does not replace the need for careful software system design. But it does let the design be modified promptly, based on early feedback, so the designer can spend less time implementing inappropriate designs. The result is increased productivity.

Enhancement. In the early stages of software development, the designer learns what the requirements really are, finalizes the design, and roughs out an implementation. In the later stages, the designer fills out the design. The quality enhancements added at this stage are those needed to transform a program into a product. These include improving the design of the user interface, changing data structures to improve performance, and improving the handling of errors and exceptions. Each step can be done incrementally, without major changes or rewrites. Again, the program can be tested between each increment. The prototype is not thrown away, it becomes the product.

Procedure. Genera's software tools operate on individual definitions instead of files. For example, the designer can compile an individual function, then immediately run the function alone to test it. Because so little is being compiled, the compilation is fast. Linking and loading occur automatically as part of compilation, providing fast interaction for

developing the design, enhancing quality, and fixing bugs.

Traditional environments require that entire files or groups of files be compiled. Therefore, compilation and relinking take a long time, and designers batch the changes between each recompile. This forces designers to keep track of too many details mentally, with the potential for losing the train of thought. In addition, they might be tempted to do other work — or get coffee — while waiting for the compilation, impairing concentration. Because designers are the limiting resource, productivity is not enhanced by such environments.

Careful evolution of a design from one stage to the next need not result in much vestigial code (also known as "temporary hacks and kludges"). This issue is as much cultural as technical; some programmers remove superseded functionality, others do not.

Open system

Genera places no barriers between the system and the user; it has no protected core or kernel. Everything (functions and data) is in the same virtual memory — any function written by the user is just as good as a function provided by the system. System and user programs are written in the same language using the same development tools, call the same facilities, and are invoked in the same way. Figure 2 illustrates the difference between open and closed systems.

Software designers can approach an open architecture three ways:

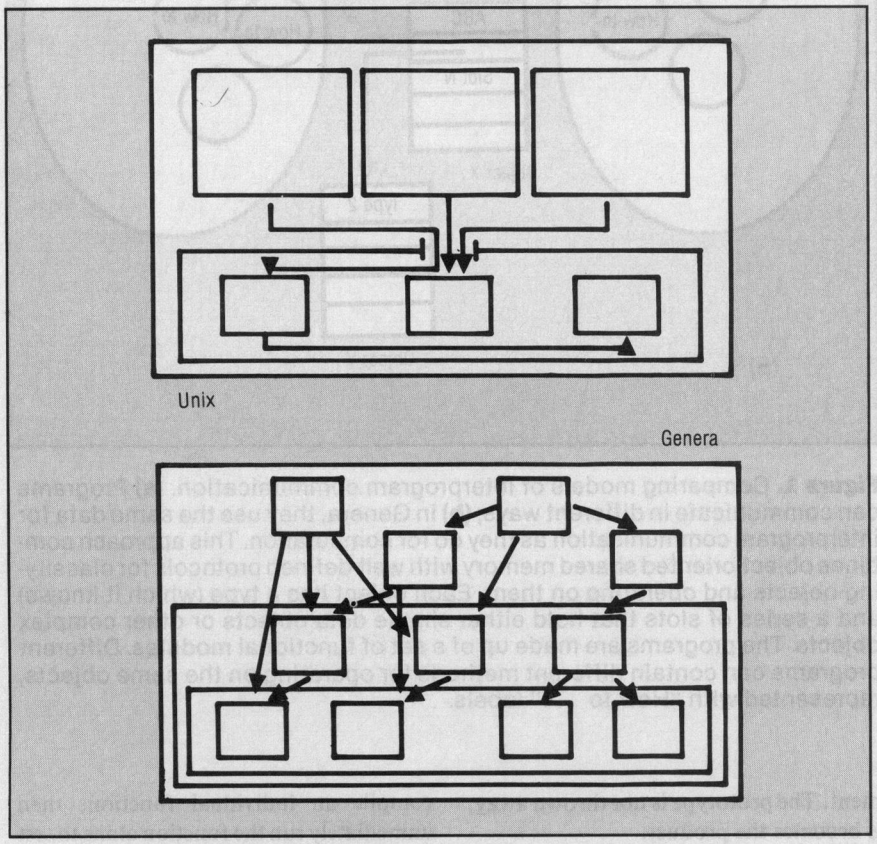


Figure 2. Comparing closed and open systems. The components of a closed system like Unix are not visible to anything outside the kernel. In an open system like Genera, the facilities available are designed at different levels of abstraction but can be called from any layer.

1. Use what's there. The system provides a huge set of capabilities, from high-level substrates to very low-level primitives, all equally accessible.

2. Use what's almost there. Designers can take what the system provides and extend it by adding some operations or exploiting some hooks. The Flavors object-oriented programming system was designed specifically to meet the goals of abstract data integration and extensibility.[4] With Flavors, several specialized versions of an existing data type can be created without interfering with each other or with the original type. Also, designers can create specialized versions of program functions without having to understand all of the original program or modify its source code.

3. Replace what's there. Major parts of the system can be (and have been) replaced by users interested in exploring research issues or enhancing performance in special situations. The open but layered structure allows higher level interfaces to be bypassed in favor of lower level interfaces. As a very simple example, a programmer can provide an alternate file system, using documented disk-access primitives.

How can a system work reliably without a protected kernel? Why don't multiple programs sharing the single virtual address space interfere with each other?

First, data-level integration (the memory contains objects, not just bits) ensures that programs agree on the semantics of shared objects.

Second, hardware support prevents the most common ways buggy programs damage other programs in traditional systems. Runtime array-bounds checking prevents wild stores. Data-type checking detects both uninitialized pointers and data structures not in the assumed format. Automatic reclamation of unused storage avoids manually reclaiming storage that might still be in use.

Finally, a program can extend or replace part of Genera for its own purposes without interfering with the operation of other programs. The principal mechanisms for extension are per-process dynamic state and flavor specialization (described above). Replacing part of the open system is usually done by bypassing the original

(rather than removing or changing it) so it remains available to other programs.

Reusability

Reusing software dramatically reduces development time. A large software product has many pieces, many of which have been built before for some other purpose. Building a new product from existing pieces reduces the size of the new job.

In an open architecture, many modules are available that are both reusable and designed to fine-tune new applications. For example, Genera includes modules that implement sorting and hashing. Every system function that needs these operations uses these modules and any user program would, too, because they are all documented.

Some aspects of Genera promote reusability:

• Visibility. Every function in the system is always available in virtual memory. There are no loadable program libraries that can bring special-purpose or potentially conflicting routines along with the ones the programmer needs. Several facilities let programmers explore the space of available functions to find ones that are good candidates for reuse. For example, the Find Symbol command lists any definitions in the system whose names contain a particular string; another command does the same kind of search of the on-line manual.

• Direct access. Customers can purchase source code for many system facilities, and various commands link compiled definitions to their source locations, giving the designer direct access to relevant reusable code.

• Generality. System modules are designed to be reusable. For example, for a hash-table module, (1) the keys and values can be any kind of data object; (2) you can provide a function to compute the hash code; (3) it automatically selects an appropriate storage-and-access algorithm depending on the table size; and (4) it lets you specify if operations on the hash table should be locked so they are atomic to other processes. Lisp's provisions for run-time type-checking and procedures as values make this possible.

Genera facilitates and encourages widespread use of reusable software. There is nothing second-class about user code — a reusable module written by any developer behaves exactly like a reusable module provided by the system.

Dynamic linking in a single, shared virtual address space makes it easy for one module to call another and allows a single copy of a module in virtual memory to be shared by many calling programs. For example, there is only one copy of the hash-table module in virtual memory even though many unrelated applications use it.

Ironically, one barrier to the reuse of modules in practice is the ease of developing facilities, even sophisticated ones. Programmers often find it easier to reinvent than to look for the relevant reusable code.

Genera's design philosophy holds that no action of the software should be hidden from the designer.

Reusable modules are not limited to low-level facilities such as sorting and hashing. An example of a high-level reuse is the extensibility paradigm of the user-interface management system. User input and output interactions are handled by a system of *presentation types*. For example, when a host name is displayed on the screen, it is not just characters, but a *presentation* of the internal object representing that host.[5]

The user operates on the object by pointing the mouse at the presentation. A function collects input from a user, not by reading a series of characters but by accepting an object of a specified type, such as a host. The system takes care of reading characters, allowing the user to point with the mouse at an acceptable presentation, checking for errors, completing partial input, and offering context-sensitive help. To build a new application interface, a designer implements a set of application-specific presentation types and associated commands that form a seamless extension of the predefined presentation types.

Extensibility

Genera's tailoring facilities, a benefit of the open architecture, are based on the belief that a happy user is a more productive user. The more an individual's work style and preferences can be supported, the more productive we expect that person to be.

In Genera, users are free to extend the system with as many new functions, commands, and utilities as they want. This capability currently is supported with simple login and application-specific init files that are loaded at login or program start-up time.

We recognize that discussions of extensions and customizing must take maintenance into account, so we provide several strategies for managing customizations:

• Most system facilities implement different user styles with settable variables. For example, when a user tries to edit a file that doesn't exist, should the system create a buffer for a new file or refuse the request? There isn't one right answer, and different users feel very strongly that their answer is the correct one. In this situation, Genera would offer both facilities under control of a variable.

• Some facilities provide hooks that let users specify custom procedures that run when a particular function is invoked. For example, many users attach a routine that deletes duplicate messages to the hook for incoming mail.

• One facility lets users wrap their own functionality around a particular system function by telling it to do "something extra" in addition to its actual definition. The something extra (which we call advice) can be done before, after, or around what the function is defined to do. The advice and the definition are independent, so changing either one does not interfere with the other.

Self-revealing

Along with the open architecture, Genera's design philosophy holds that no action of the software should be hidden from the designer. This has led both to a principle of "maximally informative interfaces" and to a commitment to supply

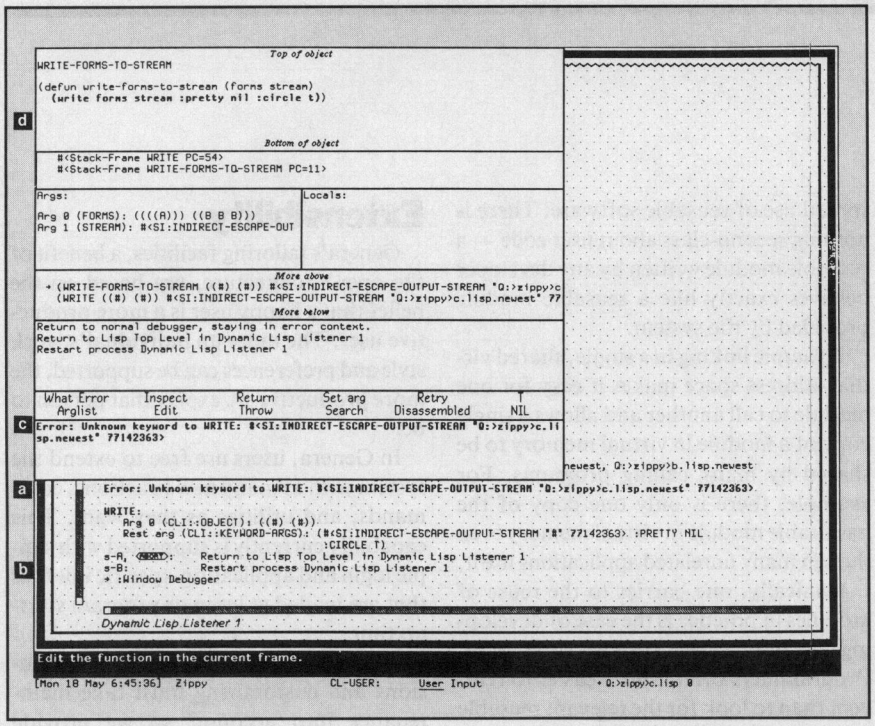


Figure 3. (a) The user, testing a Lisp function, encounters an Unknown Keyword error and **(b)** enters a command to invoke the Window Debugger **(c)** which starts the debugger process with the error message visible. **(d)** The source code for the erring function appears with the erring line highlighted. The user can look at stack frames, arguments, local variables, and any other aspect of program state. The user can not only examine the state, but can also modify it using source-language code. In this case, the user decided to fix the function containing the bad call to [write] and so asked to edit the function [write-forms-to-stream] by clicking on the Edit command.

feedback, support exploration, and supply documentation.

Without any user action at all, the following status information is always visible at the bottom of a screen, as shown in Figures 3-5: date and time; running process name; current package (the context in which symbols are read); state of the current process; progress of file writing, compilation, or other lengthy computations; and function of each mouse button, based on the position of the mouse and the state of the program.

In addition, the Peek utility displays the states of all processes, windows, network connections, servers, network hosts, virtual memory areas, and so on.

Several utilities let users examine data structures, either statically or dynamically. The Flavor Examiner shows definitions, inheritance relationships, and methods in the object-oriented programming system. The Inspector lets users see or manipulate any aspect of the current state of a complex data object.

Most importantly, users can always suspend any computation at any time and use the debugger to inspect the data objects, the call stack, and other aspects of the current computation. Again, this capability is made possible by data-level integration — the debugger has access to the objects and state of the suspended process.

Several kinds of context-sensitive documentation are available. Any time a programmer is typing a function name, a single-key command displays either its arguments, using information that it gets from the compiler, or its full reference documentation, supplied by Document Examiner (the utility that manages on-line documentation). Mouse-button documentation is generated automatically by the user-interface management system. As a user moves the mouse across the screen, the mouse documentation line changes with the nature of the object beneath.

Multiple processes, multiple activities

At the software level, Genera supports multiple independent processes in a single address space with a simple priority scheduler.

Users develop a rich model of the different activities they engage in and the methods for carrying out different activities at the same time. As far as users are concerned, some activities occur in parallel, such as editing a file while compiling another file and loading incoming mail in the background. In other cases, the users view what they are doing as moving from activity to activity, dropping one to resume another.

Genera supports this mental view by concurrently maintaining the state of any number of processes. The user doesn't exit one and return to command level to start or resume another, but instead just selects the process that will implement the next activity. The process starts up again at exactly the same point the user left it.

This structure of multiple independent processes is a strong foundation for the activities involved in software development. The screen images in Figures 3-5 show a simple example of how a user approaches debugging a problem, switching from one process to another and finally resuming a process that had previously entered the debugger. The repairs are made with the normal source language, not some error-prone, numeric, patch-up technique. In very large computations, a user can gain a lot of productivity from the ability to repair the state of a running program rather than having to restart it.

Genera evaluation

Designers moving from other environments to Genera report gains in both personal productivity and job satisfaction. We believe this is due to

- data-level integration, which permits the construction of integrated, communicating tools, directly resulting in less mental overhead;
- reusability, which lets designers start from a much higher base — the major effort goes into understanding the application requirements instead of building subprimitives to support the tasks; and

• the open architecture, which lets designers explore an idea as far as their own creativity takes them, rather than as far as the system's designers will let them go.

Genera's approach is not without hazards. Specifically, we have encountered four major classes of problems, one unique to an open system, the others common to all software environments but exacerbated by the open architecture.

1. Open systems have no "keep out" signs. An open software architecture is a boon to those who understand its risks. But without a system/user distinction to warn them off, users often choose to solve a problem using primitives instead of facilities more appropriate to their problem.

2. In an open system, documentation typically bears the responsibility for describing the levels of abstraction and the protocols for using appropriate levels of functionality for a problem. Writers have a significant burden to make the documentation of internal functions as accessible as their code.

3. The system software has been designed with calling interfaces at various levels of abstraction. Unfortunately, the boundaries between the layers of abstraction are mostly invisible, with no formal definition. Both customers and system developers have difficulty choosing a level of abstraction for a problem and staying in its boundaries.

4. When all system levels are open, compatibility becomes a difficult philosophical problem. Users want the software to evolve and improve, but at the same time they want their own code, based on changes to its internals, to keep working. Because of the extremely high level of integration, it generally isn't possible to support both old and new approaches in parallel. We have no magic solution to these inherently conflicting interests.

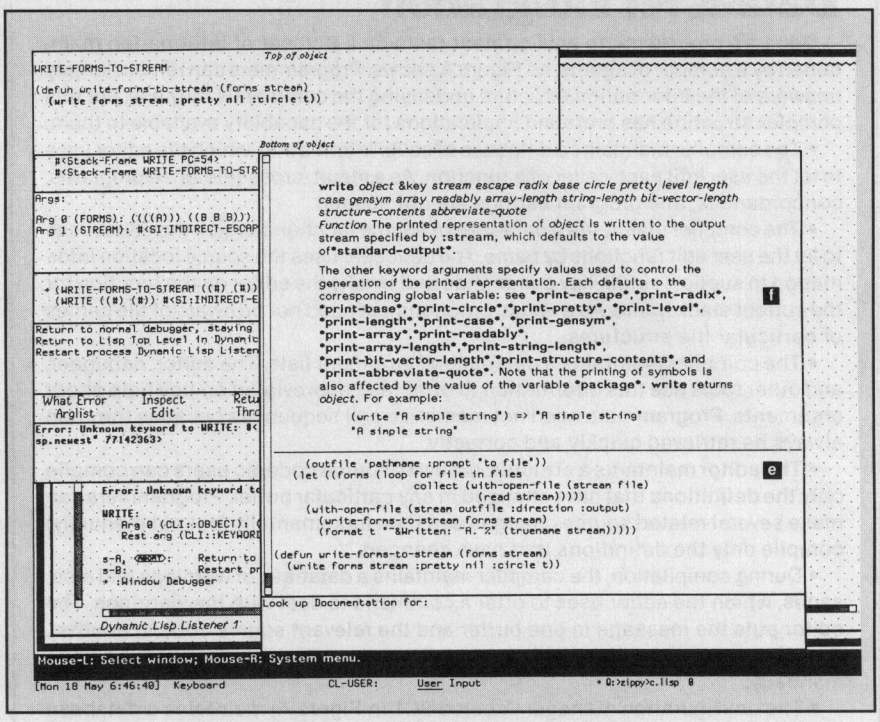


Figure 4. (e) Clicking the Edit command calls the editor process, with the code for [write-forms-to-stream] showing in the buffer. **(f)** Because the user is unsure how to fix the problem, the user invokes the on-line documentation for the [write] function. Satisfied with the explanation, the user changes the erring function, recompiles it, and returns directly to the Lisp Listener window.

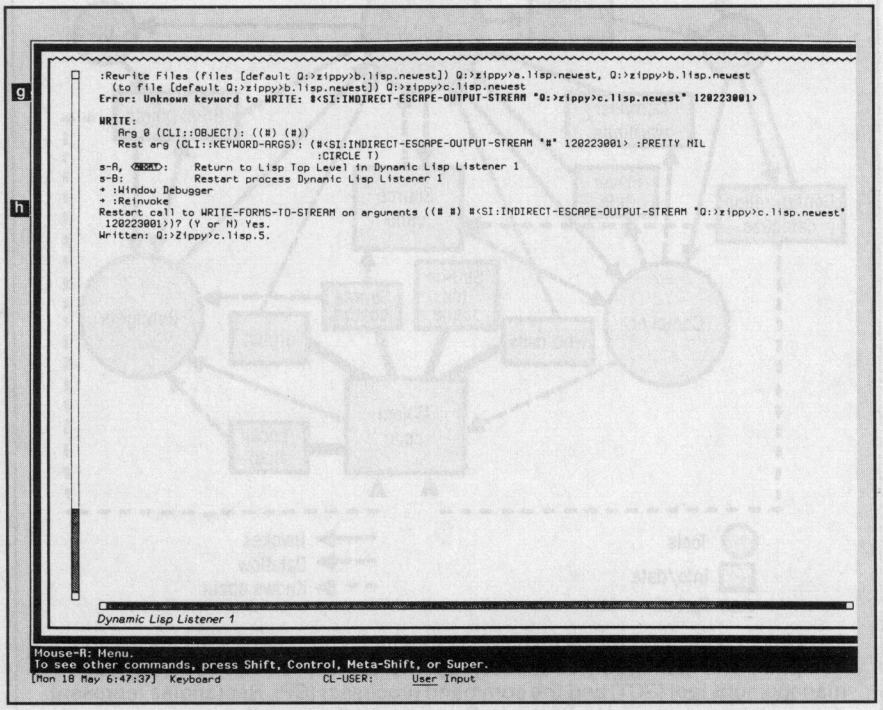


Figure 5. (g) The computation that encountered the error is still in its original state. **(h)** Having already fixed the error, the user reinvokes the function that had the problem and resumes the computation using the corrected function. Any erroneous invocations further up the call stack are repaired similarly.

Future directions

We anticipate several major new directions for Genera, in areas that may roughly be called databases, version control, and user interface. In reality, these are extensions of concepts already embodied in the system.

• Databases. Currently, data that per-

Data-level integration

Genera's programming environment rests on a rich set of information maintained by a number of agents, as Figure A shows. Programmers can remain largely unaware of the information structure underlying the commands they use, but this complex structure has profound implications for the capability available to them:

- The compiler maintains a database of caller information, which the editor uses to let the user edit each caller of a function. As a result, cross-reference programs, concordances, and program listings are not needed.
- The compiler maintains a database of source locations, which the editor uses to let the user edit functions by name. The debugger uses the source location information to support single-key commands that invoke the editor on the function for the current stack frame. As a result, programmers need not be aware of file names or particular file structures.
- The compiler maintains a database of argument lists. The editor, debugger, and other tools use this information to offer fast, abbreviated on-line help about arguments. Programmers need not memorize call sequences because they can always be retrieved quickly and correctly.
- The editor maintains a structured view of source code, so users can compile only the definitions that have changed in any particular buffer. Programmers can make several related source changes and then incrementally (but not manually) compile only the definitions that have changed.
- During compilation, the compiler maintains a database of warnings and messages, which the editor uses to offer a command to deal with the warnings. The editor puts the message in one buffer and the relevant source code in another. Programmers need not write down error messages or find the relevant definitions manually.
- The configuration manager (labeled SCT in Figure A) maintains a database both of the file names and file versions that constitute any software system and of the various compile- and load-time dependencies between the files. This database is used in full-system compilations, in incremental patching, in system distribution, and so on. As a result, programmers are freed from manual operations and costly errors in shipping software products. In addition, many operations can be performed on a system without the programmer needing to remember any of the files that it contains.

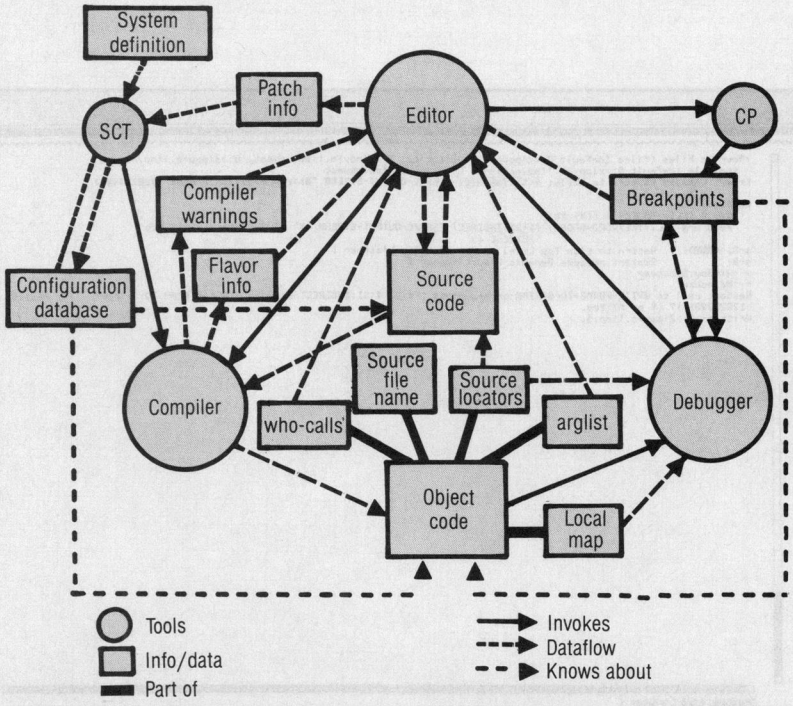


Figure A. The Genera programming environment. The primary agents are the editor, compiler, and debugger; secondary agents are the software configuration-management tool (SCT) and the command processor (CP). Rectangles represent data maintained or used by the tools. For example, the compiler uses source code maintained by the editor to produce object code, which consists of the code plus auxiliary information. This information, maintained incrementally by the compiler, is used by both the debugger and editor.

sist between sessions are stored in files that act just like their counterparts in conventional systems. This is an unnatural separation between usage format and storage format, requiring many programs to have modules for parsing and printing files to get their application-specific data in and out. We could remove this burden by using an object-oriented database to introduce persistent data-level integration.

- Version control. Currently, Genera uses a file-based system to manage multiple versions of software sources. A database of code objects would provide more flexible and powerful facilities for controlling and auditing changes.
- User interfaces. Presentation types act as a formal mediator between users and a program's data. Currently, an application program determines the appearance of presentation objects. By adding the concept of *viewspecs*, both user and program could be given independent and dynamic control over such aspects of the appearance of data objects as the level of detail displayed or the language used.

Genera's integration of diverse development activities is made possible by sharing virtual memory data objects among all processes in a machine. An open software architecture, without the conventional user/system distinction, allows exceptional flexibility and code reusability. Specialized high-level substrates for areas like user interaction also foster reusability and substantially reduce conventional design burdens on developers.

We believe the future lies in extending data-level integration beyond the realm of normal program-created data. -⊙-

Acknowledgments

Many people have contributed to the Genera software system and to our continually increasing appreciation of the methodology it supports. We thank our coworkers at Symbolics, past and present, for their vision, hard work, and unfailing enthusiasm. We thank our users for their patience and constructive feedback.

References

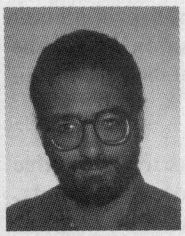
1. D.A. Moon, "Symbolics Architecture," *Computer*, Jan. 1987, pp. 43-52.
2. B. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, N.J., 1981.
3. F.P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, April 1987, pp. 10-20.
4. E.C. Ciccarelli, "Presentation-Based User Interfaces," MIT AI Tech. Report 794, Massachusetts Inst. of Technology, Cambridge, Mass., 1984.

Janet H. Walker is a principal member of the technical staff at Symbolics, Inc. Her research interests include user interfaces for software and document development environments.

Walker received a BSc from Carleton University (Canada) and the AM and PhD in cognitive psychology from the University of Illinois at Urbana-Champaign. She is a member of ACM and the Computer Society of the IEEE.

David A. Moon is a technical director and founder of Symbolics, Inc. He has been a hardware designer, microprogrammer, and writer of manuals at Symbolics. His interests include advanced software development and architectures for symbolic processing.

Moon received a BS in mathematics from the Massachusetts Institute of Technology.



Daniel L. Weinreb is a technical director and founder of Symbolics, Inc. Previously, he was director of software development at Symbolics. His research interests include software development environments and object-oriented databases.

Weinreb received the BS in computer science and engineering from the Massachusetts Institute of Technology.

Mike McMahon is a technical director and founder of Symbolics, Inc. His research interests include advanced user-interface tool kits and next-generation software development environments.

McMahon declines to name academic or professional affiliations.

Questions about this article can be addressed to the authors at Symbolics, Inc., 11 Cambridge Center, Cambridge, MA 02142.