SOME RESEARCH DIRECTIONS FOR LARGE-SCALE SOFTWARE DEVELOPMENT

David G. Belanger, G. David Bergland, and Mike Wish

David G. Belanger, G. David Bergland, *and* ***Mike Wish*** *are with AT&T Bell Laboratories in Murray Hill, New Jersey. They are responsible for research and technology transfer of tools and concepts that give productivity and quality leadership to AT&T in the development of software products. Mr. Belanger, head of the Advanced Software Department, does research on software productivity tools and concepts, emphasizing program generation and database management. He joined the company in 1979, and has a B.S. from Union College and an M.S. and Ph.D. from Case Western Reserve University, all in mathematics. Mr. Bergland, head of the Digital Systems Research Department, is involved in research in software engineering, specification languages, programming environments, and photonic switching*

(continued on page 92)

We describe a collection of research efforts aimed at increasing software productivity and quality. Some projects seek to understand and improve the process of efficiently creating, controlling, and assembling the products of software development (i.e., software and documentation). In these areas, the problems relate to the complexity of managing communications among a software product's elements (including people) and of managing many versions of the product, each at a different stage of its life cycle. To address these goals, several research projects provide more supportive languages and programming environments for building pieces of a product, or make available more powerful software development environments for integrating these units into a system or product. Other approaches, aimed at tools and concepts that change prevailing software life cycles, include automating the creation of software. Results of research in this direction suggest potential for dramatic improvements in productivity and quality.

Introduction

Software—all the associated documents and code—often absorbs more than 80 percent of the development time and cost of major computer-based systems. Therefore, basic improvements in software technology can dramatically affect the software development process.

This paper provides a guided tour through some of the research at AT&T Bell Laboratories that addresses opportunities for improving large-scale software development. (Panel 1 defines acronyms, and Panel 2 identifies software tools and languages.) The goal of this research is to build the technology base for a large increase in software productivity and quality. This tour does not exhaust the wide range of those research

Panel 1. Acronyms in This Paper

AAAI	American Association for Artificial Intelligence
ACM	Association for Computing Machinery
ASCII	American Standard Code for Information Interchange
CASE	Computer-aided software engineering
COMPSAC	Computer Software and Applications Conference
FSM	finite state model
IPC	interprocess communication
SDE	software development environment
SIGPLAN	special-interest group on programming languages
SIGSOFT	special-interest group on software engineering
TMS	truth maintenance system
WYSIWYG	what you see is what you get

Panel 2. Tools and Languages

`bonsai`	a graph processor and editor
Cens	C-language environment system
`cia`	C information abstractor
`cin`	C source code interpreter
Counterpoint	a reasoning-based graphics toolkit
`cscope`	C-language code browser and editor
`dag`	tool to draw a directed acyclic graph
`grep`	tool for pattern searching
IFS	Interpretive Frame System
`inscape`	set of tools for specifying a software system's interfaces
Instress	interface specification language for `inscape`
`monk`	database-driven document formatter
`nmake`	new `make`; tool for maintaining and updating multifile programs
PAISLey	process-oriented, applicative, interpretable specification language
Pegasus	a program generator
PML	version of the ML language for the Pegasus system
`rip`	an automatic code generator
SABLE	software product administration system
`sam`	a visual editor for bit-mapped terminals
`samuel`	a visual C-language browser and editor (built on `sam`)
`tbl`	preprocessor for formatting tables
`vi`	visual editor
`watson`	automatic programming tool

efforts. Instead, it stops at a few points of particular interest, providing a glimpse of some technologies that may help revolutionize the way software is developed. In general, these experiments are among those that are mature enough to have taken on some form and may become tools for software development within five years.

Levels of Software Development Activities. Numerous problems are encountered when a software project "scales up" to a system that requires many developers and has a life cycle of many releases. When we view the technology required to develop large-scale software products, it is convenient to consider activities at three levels of complexity—product, system, and unit—each with a software development environment to support it.

The *product* level's primary concern is to support the development and maintenance of large, multirelease software products over a complete life cycle (see Panel 3). A multirelease product owes much of its complexity to parallel entities of many types that have tangled relationships. For example, programmers may be working in parallel on various parts of the system, and different releases of the system may exist in the field at the same time. Also, there are likely to be several versions of each module in the system and a variety of target environments on which the system must run. One goal of research at the product level is to reduce this complexity to nearly that of a single programmer updating a system with only one version.

The *system* level is concerned with creating a *single* release of a working system, completely tested and with associated documentation. A driving force is toward software generation of system parts and integration (reuse) of existing software into complete systems. Considerable research concentrates on the front end of the system life cycle, topics such as specification and design technologies. Various studies have indicated that 50 to 70 percent of the errors discovered during system testing result from problems with the interface that integrates the units into a completed system. So, a primary concern at the system level is to eliminate these errors.

The task of creating the individual modules of a system—often called "programming in the small"—resides at the *unit* level. Here, much of the research thrust is on more powerful, integrated support for creating, testing, and debugging units, as well as on automating the generation process. Besides software modules, the units include components of the requirements, documentation, and test scripts, and nearly all parts of the system life cycle.

As Figure 1 illustrates, many tools are available to create software units efficiently. Historically, there has been much active research in this area. Although fewer tools are available to aid the system and product activities, there is growing awareness that they are essential to further productivity gains. As a rule of thumb,[1] to advance one level in Figure 1—e.g., produce a system from a collection of units—requires three times the effort of the previous level. Thus, a multirelease product costs about nine times as much to develop as the associated units.

Research into tools that will support the product, system, and unit activities in application-software development projects typically involves building prototype software "tools" that embody the concepts of the experiment. This prototype software is often complex, but typically does not involve large teams of programmers working in parallel. Given these characteristics, we commonly use the elements of the unit development environment to build research software, just as we use them to develop applications. However, the needs of application development are much more pronounced in the system and product activities.

In the rest of this paper, we introduce a few research projects that could advance the software development process. For simplicity, the sections relate to activities in Figure 1 and the likely application of the research. But many research efforts support more than one level, and some are attempts to create integrated environments to manage all levels.

Panel 3. Versions, Releases, and Generics

The term *version* refers to one of several, usually somewhat different, copies of a part of a system. For example, a single function, X, may have been enhanced five times over the life of the system to which it belongs. It, therefore, has at least five versions; but there may be others, such as a version under current modification. Typically, when we collect versions of a product's software modules and documentation, "freeze" them as of a fixed date, and ship the collection as a product, we call it a *release* or *generic* of the product.

Product Development Environment

Here, we examine some research into problems of developing and maintaining large, multirelease software products over a complete life cycle. This requires control and evolution of all parts of the product, as well as automation of the process. Figure 2 illustrates part of that process for a single release of a software system.

At this level, one must coordinate control of parallel lines of development with communication about the software product. At any given time, programmers will be working on different modules, while maintainers, testers, documenters, and others will be working on their respective parts of the system. Many versions of the system—usually for different machines—will also exist and be in use concurrently, versions subject to maintenance changes, future versions, and on and on. The problem is similar to trying to assemble several copies of a puzzle while various contributors remove pieces at will from any copy, change their color and shape, and then replace them.

Even the smallest change to a major, multirelease system is a complex undertaking. For example, to find and fix a bug, one must at least:

- Determine what versions of each software module were used to create that release.
- Try to recreate the release exactly as it existed previously.
- Isolate and repair the faulty code.
- Determine if other programmers are changing parts of the same release.

- Rebuild and test the fixed release.
- Log changes and reasons for them.
- Perhaps fix later versions of the system likely to be affected by the changes.

The rest of this section discusses research on a prototype operating system and a software database that addresses some problems and needs for multirelease products. The research entails collecting, retaining, and manipulating:

- Information about the objects (e.g., programs or documents) associated with the product
- Relationships among these objects
- Rules that govern the use and generation of the objects (e.g., for generating the executable form of the product).

Conceptually, this forms a software information system that will support decisions and automate the process of putting together software products. These efforts, like much of the other research at this level, are oriented toward managing and reducing complexity.

We then discuss new visualization aids that are appropriate for the product, system, and unit levels.

An Advanced Software Development Environment. Currently, researchers are experimenting with a prototype enhancement[2] of the UNIX® operating system that provides enriched file (object) typing and object relationship mechanisms. Their goal is to provide a perspective that lets a programmer view and operate on the software system as his or her own. In this environment, the standard view is that of one programmer working on a large, one-version system.

The concept of *versions of an object* is embedded in the file system as a data type that supports all current types of files and directories. In addition, the concept of a *continuation directory* is provided as a relationship among objects (e.g., files, directories, versions). A continuation directory generalizes the concept of a *viewpath*[3] for objects that an individual controls, but with a "view" to the remainder of the system. A viewpath is a model that overlays different versions of objects in such a way that the user sees only the "top" of a stack of objects that have the same name. This approach extends the original model by providing transparency across all tools, file systems, and nodes on networked computers.

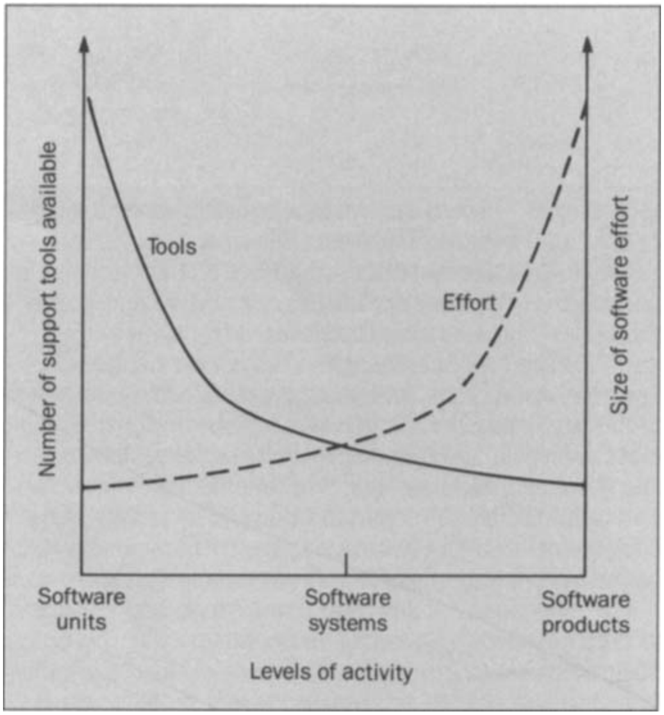


Figure 1. Support tools available and effort required as software projects *scale up*. Units, systems, and products refer to software's levels of complexity.

In Figure 3, two programmers—Dick and Mary—are modifying a system that includes objects (e.g., software, documents), such as A, B, and C. Each object has several versions. Here, red represents the latest "official" release of the software product. A system tester, Tom, views the current (green) version of the upcoming release that includes a new object, D. Because module B is unchanged, Tom also sees the official (red) version of B. Dick, who is fixing module A in the previous release, has a copy of the module (blue) in his private workspace.But Mary is modifying her own private version of the new release object D (yellow).

In this environment, which is a variation of viewpathing with the transparent addition of versioned objects, Tom views the system as the test version (green) followed by the official version (red). [That is, the system he will test consists of all test (green) objects plus those official (red) objects that do not have a test (green) copy.] But Dick views the system as his private workspace (blue) followed by the official (red) version, and Mary views it as her private workspace (yellow) followed by system test (green) and official (red) versions.

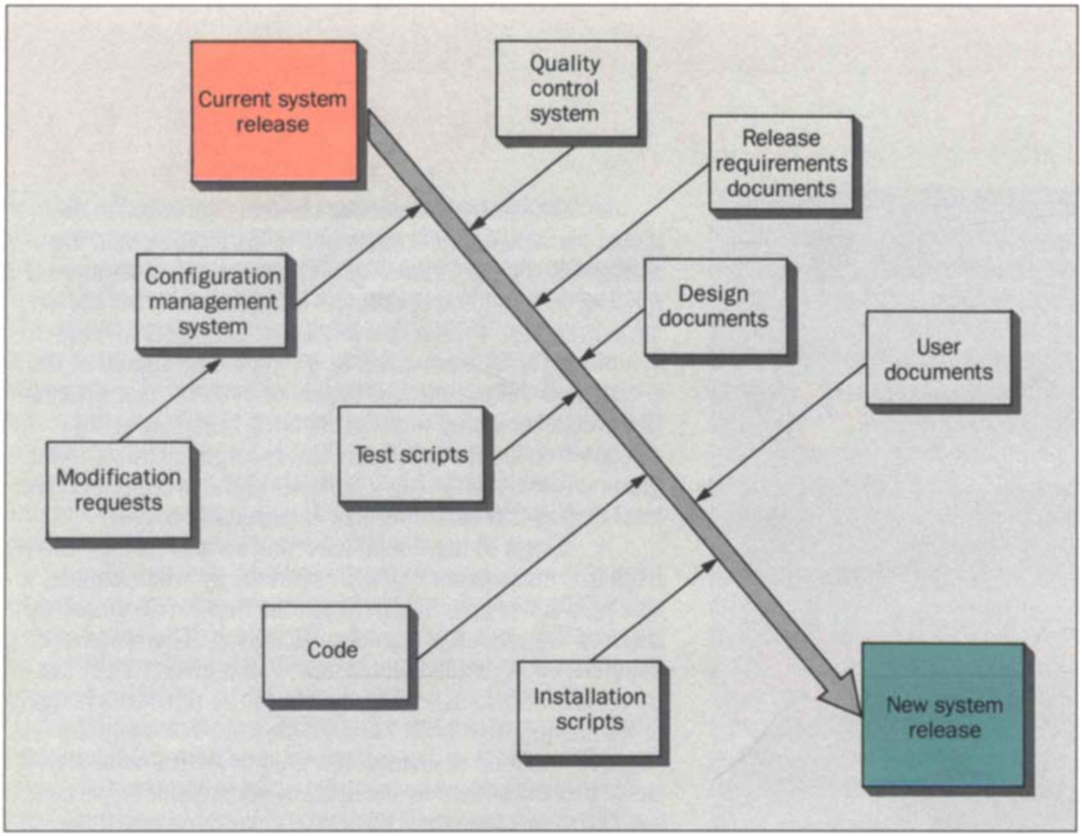


Figure 2. Control and evolution process for a single release of a system.

The prototype makes all this functionality transparent to developers (unless they need to see it; for example, to compare successive versions of an object). It manages all types of objects (not merely ASCII text files) and, perhaps most important, simplifies adding new tools. Some tools currently provide part of this functionality for the build process (principally `nmake` combined with *SABLE*[4]), but not for other software processes.

Because these relationships are embedded in this operating system, software developers use their favorite tools without modification. Standard UNIX system tools, such as `vi` or `grep`, and newer software, such as tools for static analysis (e.g., `cia`, the C information abstractor,[5] or `samuel`, a C-code browser and editor), are used as if this were a single system with one version.

This prototype now provides a basic environment for experimenting with a variety of tools. Some are user-level tools for convenient access to the system's functionality—for example, a version control system to:

- Control adding and deleting versions of an object.
- Log the information on reasons for changes.
- Manage other required data on object versions.

Others are current research prototype or standard development tools that, when they use this prototype operating system, have access to many more views of a system.

The capability that this prototype operating system provides has already significantly increased the power to understand the structure and internal relationships in an evolving system. For example, we can use `cia`, described next, to map the differences in a system's call graph that are caused by the introduction of a new release. (A call graph shows what functions call each function in the system.) With viewpathing, we can show this difference dynamically as the new release is being built.

Further research will investigate the use of a variety of programs that automatically execute, based on activity in the underlying system. These programs are likely to take on much more intelligence and provide automation. For example, when a new version of an object is created, they may warn other users of related objects or update relationships. We can easily imagine programs that initiate tasks, such as partial system build or selective regression test, when appropriately stimulated. They might even try to remerge automatically separate branches

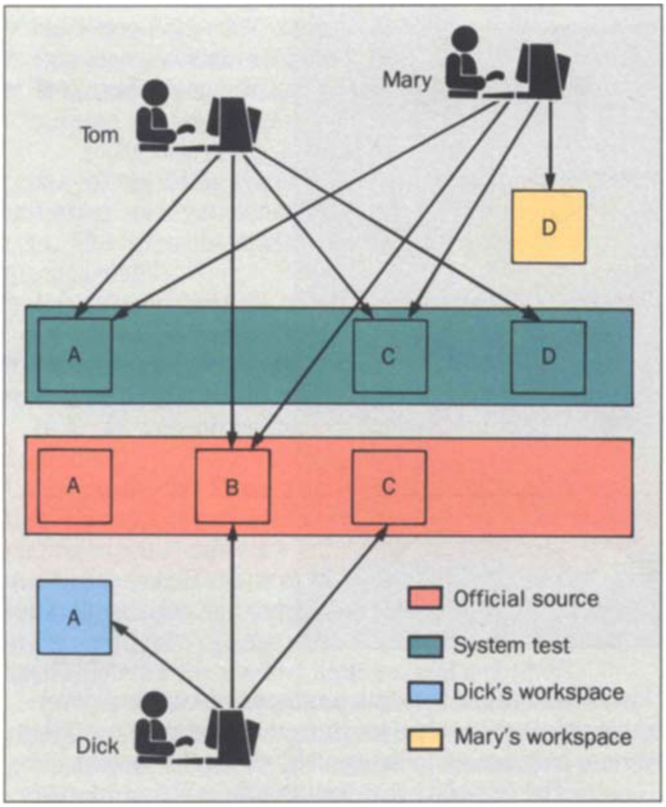


Figure 3. Viewpaths and versions. Mary and Dick are programmers working on different versions of system modules; Tom is testing the "next" release of the system. A, B, C, and D represent objects (software, documentation).

of an object's version tree, addressing the common problem of managing reused objects that have been modified in many ways.

A Software Database. Another experiment investigates ways for a software system to derive structural information so it can analyze and restructure itself. The initial prototype, an enhanced version of `cia`,[5] collects information about the static structure of software systems written in C language and stores the data in a relational database. One aim is to automate system actions based on analysis of the database.

The current database contains information on global variables, functions, macros, data types, and files, along with their relationships. The database is structured for fast, incremental updating—a requirement for use on large systems. We can derive all that data by analyzing the syntax and static semantics (e.g., type definitions) of the source code. For a small program or system, that information would be stored in programmers' minds or easily displayed on a sheet of paper. But in large systems, a programmer deals with only a fraction of the system, and the total display can run to tens or hundreds of pages.

Some of the useful information that can be derived from this environment is well understood—for example, a system's call graph. Which functions depend on global variables or are reused is also easy to derive. The system's dependency on transitive closure with a given object can be calculated and used to understand the effect of changing or eliminating that object and its closure. For example, suppose we need to change the type of a parameter that is passed to a function during system maintenance. We can use the dependency calculation to determine where to change the code.

Currently, the research addresses four issues:

- Scaling up to very large systems. This includes issues such as incremental build of the database and management of parallelism. As `nmake` evolves, it will become a critical tool in this process.
- Deriving software metrics based on the database to support decisions such as system restructuring. Clearly, this requires robust metrics to serve as objective functions for defining improvements or optimizations.
- Extending this database to objects outside the source code. Examples might be test scripts, design information, semantic information about some of the objects, or dynamic information based on software execution.
- Extending these techniques to other languages, particularly object-oriented languages such as C++,[6] and languages that support parallel programming such as concurrent C++.[7] This will introduce new problems in describing the network of software objects because the system will have to understand the object hierarchy (in the language sense).

Through analysis of the data gathered and visual pattern recognition, avenues will be opened for comprehending and improving the structure of large systems.

Aids for Visualization. We are on the verge of major advances in understanding large systems through visualization of patterns. In the past, our software developers rarely used color or graphics because software projects were naturally intractable and multidimensional. But some new work suggests that visualization can be a powerful ally to understanding an evolving system's structure and operation and identifying opportunities for improvements. Another component of this developing technology is the ability to interact with graphic displays.

To understand a large system's static and dynamic aspects, we are using new interactive techniques to display directed graphs. For example, `dag` (draw a graph)[8] uses a variety of optimizations and heuristics to convert a list of a directed graph's node adjacencies into a display of the graph. (Node adjacency can identify the functions that call a particular function and any functions that it, in turn, calls.) To generate displays, `dag` uses some newly developed algorithms, based on the network simplex algorithm, that are optimal in length of edges and have good heuristics for minimizing edge crossings. As a result, it automatically draws displays of directed graphs that visually convey as much information as possible about the underlying structure of the graph.

`Dag` provides the ability to investigate—in reasonable time, even for large systems—the many properties that a directed graph can represent. An example is exhibiting structural properties of call graphs or dependency graphs. In less than an hour, `dag` has built graphs for a system that has more than 2500 functions and 8000 connections. Typical graphs are done in less than a second.

Figure 4 is an example of a call graph for a small system of about 4000 lines of C code. Notice that this system has a function, `xyz`, that is not called within the system. By viewing the graph, we can see that `xyz` and part of the subtree it calls can be removed. On the other hand, some functions that are part of the subtree must not be removed because they support functions other than `xyz`. Finally, we see some functions (e.g., `xmalloc`) that are called by many different functions and are strong candidates for reuse in other systems.

Much of the current experiment is aimed at:

- Understanding what should be presented (i.e., what information has maximum value in pattern recognition)
- How it should be presented
- What new algorithms will be needed to support it.

We are also experimenting with the use of color—for example, in graphs of static relationships—to show special relationships, reuse opportunities, or excessive dependencies. Also being examined is the use of this capability for dynamic system aspects and for visualizing data structures as part of a debug or trace mechanism.

A third component of this approach is the ability to comprehend and interact directly with the display of a graph or network. This capability allows users to manipulate, filter, and view entities represented by nodes in the graph. A prototype system, called `bonsai`, that has this capability has been built. `Bonsai`'s input is typically the output from a layout program (such as `dag`), so it completes a loop of extracting, displaying, and manipulating objects described by that information.

System Development Environment

Here, we look at a few research projects that address the system level's core problem: creating a single version of a working, tested, and documented software product. The system level includes a single pass through all phases of the traditional software life cycle—from requirements, to design, to release.

Improvements in technology for building large systems start with tools that simplify prototyping and formal specification.[9] The Interpretive Frame System (IFS)[10] is one of the earlier successes. IFS supports incremental prototyping of highly interactive systems that can be modeled as a network of tasks (frames). Another example is *PAISLey*,[11] the executable specification language oriented toward distributed, real-time systems. It can be used to describe implementation-independent models or prototypes of software systems. These models can be executed to simulate the proposed system's behavior and performance. PAISLey can also be used to model the proposed system's environment, thus providing early documentation and, eventually, a testbed for the developing system.

Of course, one of the most powerful prototyping

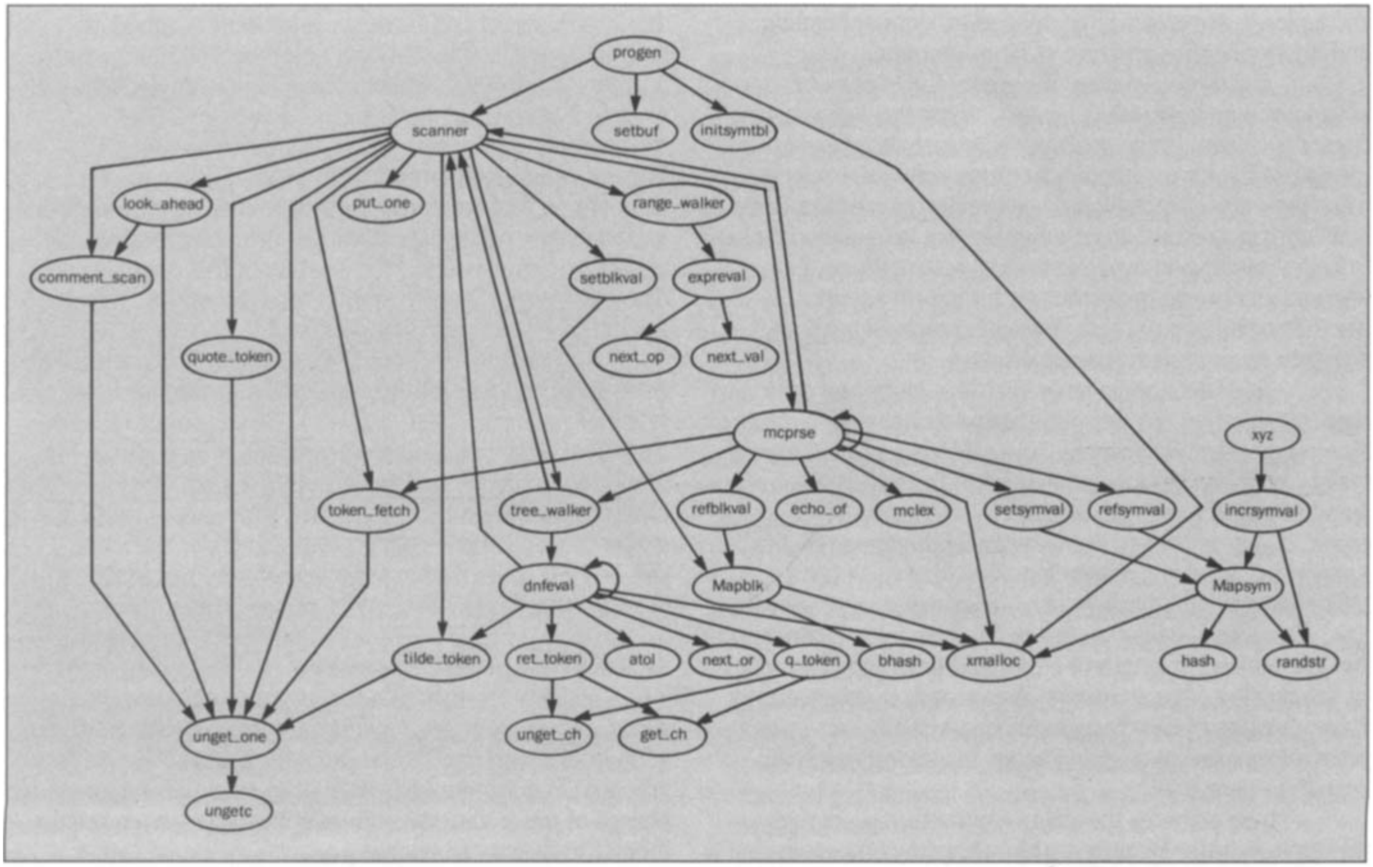


Figure 4. Call graph for a small system. Because nothing calls `xyz` and some of its subfunctions, `xyz` and those subfunctions can be eliminated. A function that many functions call, e.g., `xmalloc`, is a candidate for reuse.

toolkits is the UNIX system itself. We can use the shell as a programming language and paste together members of the rich collection of small tools.

However, individual specification and prototyping languages typically address only part of the problem for large systems. These languages must allow for easy, seamless integration with the products of other application generators and with handwritten code. This leads to other research to make the interconnections between separately built "units" more error free.

At the system level, perhaps the most important concern is the *platform*, or set of tools, on which the executing system will run. This typically includes the operating system, a database management system, and a collection of applications-oriented software such as form and menu managers or report generators. The higher the level of the platform, the less the work required to develop the running system and the more likely that techniques such as prototyping can be used effectively to shorten the system life cycle's front end.

We will give examples of three general-research directions for better ways to build large systems.

1. Seamless integration of separately built units. This

direction seeks to improve the quality of the original requirements and reduce the large amount of system work that goes into integration. For example, inscape[12] provides a way to specify and check interfaces between a system's functions as the system is being designed, thereby capturing more errors before integration.

2. Higher level platforms for key tasks. This direction hinges on finding a model of computer processing that is general enough to include many applications, and finding an appropriate language in which developers can express their applications. Research projects described here relate to two such models:
   - — Event-driven systems, such as *Pegasus/PML.*[13] Transaction processing and network operations are examples.
   - — New models of application development and user interfaces—such as *Counterpoint,*[14] a reasoning-based graphics toolkit.
3. Higher level interfaces for automatic generation of application programs. This direction is closely related to the second one in that it also requires a broad model of computer processing and an appropriate application development language. Automatic programming is illustrated by the watson and rip prototypes.[15,16]

Seamless Integration of Software Systems. Software systems are prone to specification and implementation errors. Specification errors tend to be the most expensive to correct because the feedback cycle is so long. That is, a specification error that existed on day 1 can persist through the design and implementation phases, only to be caught near the end when the system first cycles. Implementation errors occur largely because of poor communications between specifiers and developers, or poorly documented and communicated expectations.

For a smoother passage along this error-prone journey, two things seemed clear:

- We had to exercise and debug the specifications early in the cycle. This led us to consider instituting more formal procedures for capturing, specifying, and prototyping system requirements.
- We had to capture and monitor the expectations of everyone involved. This led us to create a development environment that polices the formal requirements throughout the process.

Formal interface specifications. Inscape, an experimental software development environment for system construction and evolution, consists of an integrated set of tools centered around the *constructive* use of formal interface specifications. It implements two independent concepts:

- Formal interface specifications and an underlying semantic interconnection model. (This concept is discussed below.)
- The policies, structures, and mechanisms for handling "crowd control" problems. (Crowd control involves tracking and communicating changes when many programmers work in parallel on a large, multirelease system.)

The heart of inscape is *Instress,* its interface specification language that enforces consistent use of interfaces as new components are being implemented. Empirical studies have shown[17] that nearly 68 percent of the errors that appear at system integration and test time are program-module-interface errors. Through formal specification of module interfaces, inscape seeks to eliminate 75 percent of these interface errors (or about 50 percent of the total system-integration and test errors). These specifications include details about data initialization and constraints, operation preconditions and results, and exceptions and their implications.

Instress uses Hoare's *input/output predicate* approach to enable module-interface designers to describe semantic intent for objects in the interface. Input predicates (assertions) describe the assumptions (*preconditions*) to be satisfied for an operation to execute properly; output predicates describe the results (*postcondition*) when operation is properly executed.

Hoare's approach is extended in several ways, so it applies to module interfaces used in large software systems. First, some results of an operation—such as requirements to clean up data structures or to deallocate buffers—are better described as *obligations* than postconditions. (Obligations are predicates that must eventually become true at some point after an operation's execution.) Second, operations often have multiple results (mostly

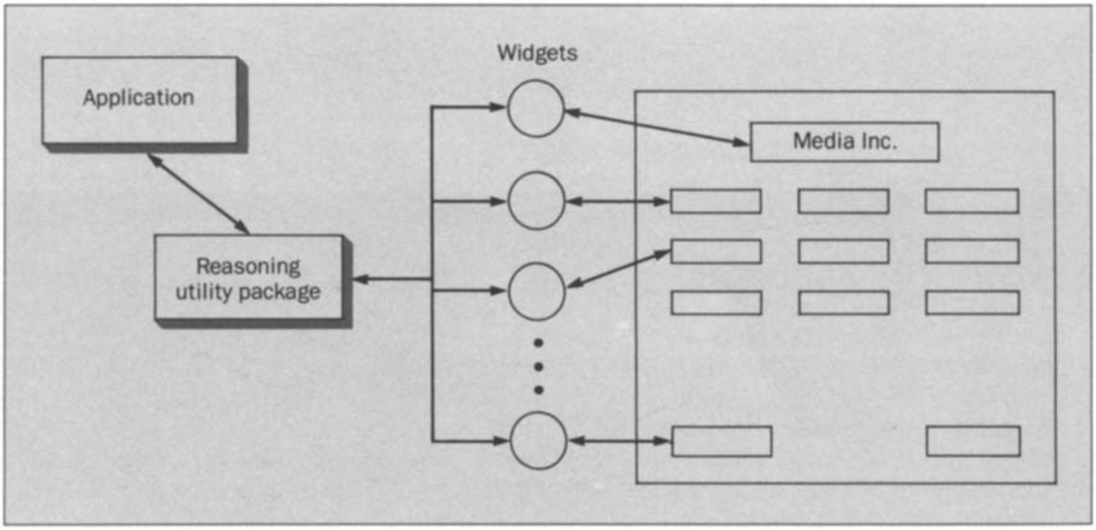


Figure 5. Schematic representation of the Counterpoint prototype, a forms package. "Widgets" is the X Windows system term for icons, clocks, and menus.

exceptions), particularly where robust, fault-tolerant software is required. Third, predicates are used to describe the properties and meaning of data types, variables, and constants.

The importance of module interface specifications in `inscape` lies in their practical utility, not their use for program verification. `Inscape` provides a shallow form of semantic-consistency checking that emphasizes pattern matching and simple deduction, not deep inferencing. In this way, `inscape` can make practical use of predicates as the basis for the underlying semantic interconnection model and constructive use of these specifications.

Higher Level Platforms. As the practice of generating code with software tools becomes more common, even for performance-constrained systems, the target languages and run-time systems can become barriers to higher level generation. An example is the need for dynamic memory management, important when programming by hand or generating an entire system, but much more so when generating pieces of a system that must later cooperate with each other. Also, demand is increasing for systems that appear integrated to users, i.e., provide an extensible variety of functions without losing performance. Finally, platforms should provide fast context switching; that will require efficient, flexible interprocess communication (IPC).

Environment generation. What motivates Pegasus research,[13] an experiment in higher level platforms, is a desire to support the generation of both programming and application environments, especially applications driven by real-time events. Two major features being explored are lightweight processes and automatic "garbage" collection (release allocated memory when it no longer is needed).

Lightweight processes require few system resources, are inexpensive to create and destroy, and put little burden on context switches. When implemented in a shared memory model, as is typical, they can provide very fast interprocess communication. With appropriate language support, the IPC bandwidth can be further enhanced to permit passing high-level data objects between processes. When the objects are read-only, the communication becomes nothing more than passing a pointer from one process to the other.

The mechanism for providing Pegasus features in a safe and coordinated way is PML, its system language. Pegasus is the run-time environment for PML that, in turn, provides the semantic model for Pegasus. PML was designed to be a good target language for program generation, and flexible enough for writing the "system" code and semantic actions that must be supplied by hand.

PML has strong support for data abstraction, which is further enhanced by PML's event mechanism. Commonly, a process may need to wait for events generated by multiple objects, modules, or processes. If the language includes event values, we can provide this wait interval by referring to the way the event is generated, such as a channel or process event, or as a timeout.

Extensible interactive environments. A second approach to building systems based on higher level platforms looks at applications that are highly interactive and, as above, are built as a collection of processes with shared data.[18] This work, a prototype graphics toolkit called Counterpoint,[14] extends the philosophy of small reusable modules to the interactive bitmap graphics world. The modules can be designed to allow application developers to customize interfaces simply, while ensuring that their applications have consistent and friendly interfaces for less experienced users.

We are using Counterpoint (Figure 5) to experiment with high-quality, extensible programming and application development environments. Eventually, it will let application developers describe—with one set of constructs—the construction of graphics objects such as icons, clocks, and menus (called widgets in the X Windows™ system); interactive screen layouts; and the sequence of interactions with the application's user. (X Windows is a trademark of the Massachusetts Institute of Technology.) The constructs are based on notions from truth maintenance systems (TMSs)[19] and constraint propagation. From an application development viewpoint, the TMS permits the use of logic to determine the flow of control between objects. To insulate the traditional host side of an application from the interactive part, communications are funneled through the TMS, which acts as the system's database. The TMS, in turn, controls and communicates with the objects.

The use of constraint networks makes it possible to run computations forward and backward, solving for unknown quantities. In such a system, a small perturbation in a node may activate the rest of the network. This model is effective for many, common layout problems found in interactive computer graphics. It will be useful for other applications, such as visual shells (e.g., the Macintosh™ computer's user interface), spreadsheets and analytic tools, form and file management, and network applications. (Macintosh is a trademark of Apple Computer.)

Automatic Programming. Both `watson`[15] and `rip`[16] automatically generate code based on a specification, an approach that raises the level at which applications are built. These prototypes address three needs in building large systems:

- Prototyping to determine requirements
- Consistency and completeness checks of the requirements as they are developed
- Eventual generation of executable code that matches the specification.

Watson. The `watson` prototype computes formal behavior specifications for process-control software—specifically telephone call control software—from informal "scenarios" that represent traces of typical system operation. As Figure 6 shows, `watson` generalizes scenarios into stimulus-response rules, uses the rules to build and refine a partial finite-state-machine model, and then audits and augments the rules to repair inconsistency and incompleteness. Finally, it produces a formal specification for the class of computations that produce the scenario, and proves compatibility with a set of domain axioms.

A particular automaton from the computations class is constructed as an executable prototype for the specification using weak temporal logic. To generate the final code, `watson` does a series of transformations on the resulting finite state model (FSM) and its state transition rules. To validate the resulting program, we can test it against the user's original scenarios.

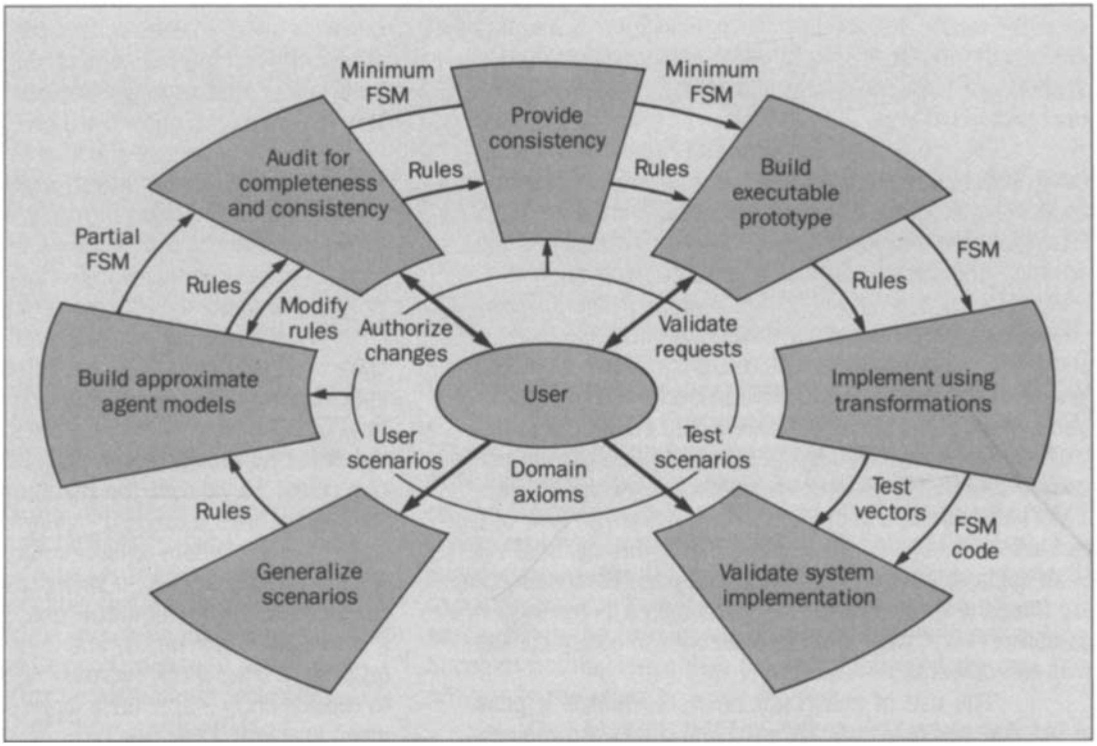
The *domain axioms* codify common-sense hardware constraints (e.g., *a phone can't ring if it is off-hook*), rules of telephone etiquette (e.g., *your phone shouldn't ring if you're not being called*), and logic (e.g., *for something to begin to be true, it can't already be true*). These axioms tend to remain fairly stable for a large class of features within a given problem domain.

Rip. The `rip` prototype gives improved fault-tolerant behavior to program modules in large software systems. Its contributions, as compared to earlier automatic programming approaches, include a rich diversity of programming knowledge in its knowledge base and support for modular system designs through formal interface definitions.

As Figure 7 shows, `rip`'s output is a robust program module that exhibits run-time error detection and damage confinement. `Rip` derives this output from a naive program module (i.e., one with no error legs), a specification of "interesting" failure modes, and formal module-interface specifications similar to those that `inscape` could provide.

`Rip` lets a programmer specify and fine-tune an error-handling strategy for a particular application, and then apply this strategy mechanically to every module in the application. During a project's early phases, developers can specify maximum error checking to simplify integration testing. Later, they can gradually relax the error-

Figure 6. A system's scenarios as produced by the `watson` prototype, an automatic programming tool.



checking constraints to achieve real-time performance goals. Ideally, one could adjust a system's error-handling strategy while it operates, but `rip` does not yet support this. Currently, we must freeze an error strategy into code at compile-time.

In effect, `rip` provides an alternate way to specify error strategies. Because the strategies can come from a library of known, trusted techniques and are machine independent, this approach aids the cause of program reuse.

Unit Development Environment

The development of units of software, documentation, and tests is the "atomic" task in building a large software system. Typically, a large system will be decomposed through several layers, the last being the individual (atomic) units that must be coded. Ideally, and occasionally in practice, the basic unit is available in an existing library or can be created by program generators.[20] This should be true for documents as well as software, especially at the generation level.

Activities at the unit level are more diverse than at the system and product levels. Traditionally, the unit level has been the source of the most progress in software development research. Today, there is a remarkable collection of tools to create, debug, and improve individual programs in environments such as the UNIX system. The projects we describe are among those that provide—without losing the powerful array of tools currently available—more integrated support for unit development and, often, parts of the larger system. The aim of this research is to create environments that combine the highly supportive characteristics of interpretive systems with those needed for a multiperson project that uses compiled languages such as C and C++.

An Integrated Environment for C Programming. One direction of research at this level is directed toward a fast,

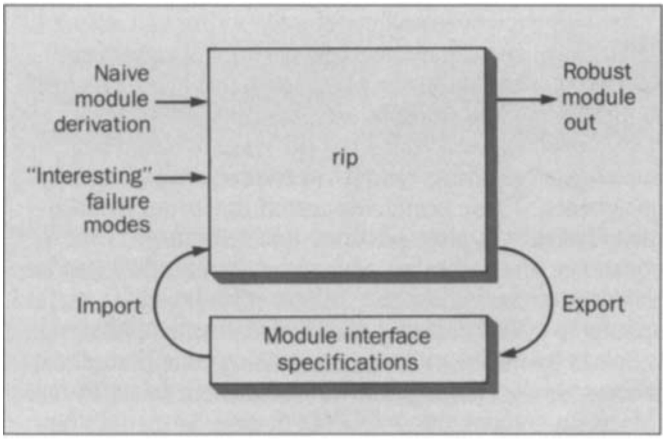


Figure 7. A view of `rip`, an automatic code generator.

interactive environment for creating, editing, browsing, executing, debugging, testing, and maintaining C-language programs. In current environments (e.g., UNIX systems), individual tools serve many of these functions. The aim of this work, called *Cens* (for C environment system), is to create software with more functionality that:

- Works together in a flexible and integrated way
- Preserves the ability to use individual tools as needed.

At the heart of Cens is a C source-code interpreter, `cin`, supported by an incremental loader that links object files and libraries to interpreted routines. It is in `cin` that the C functions are created and initially tested. `Cin` has an incremental parser that accepts and checks full C syntax, while providing complete breakpoint and trace facilities. These facilities let a programmer create code, test it immediately, and run in debugging mode. Perhaps more important for large systems, a programmer can use `cin` during maintenance to learn how existing modules operate by running them in interpreted mode, while the rest of the system runs in compiled code. Finally, `cin` supports software reuse because new routines can be individually created, tested, and efficiently integrated with the rest of a working system.

When scaling up to work on large systems, performance is a critical issue. An incremental loader is used with `cin` to minimize run time by limiting the interpreted portion to routines under development, while executing the bulk of the program as compiled code. For `cin`, current benchmarks indicate interpreted code is 35 times slower than compiled code. Because a programmer can use any combination of source and object code, execution time and memory requirements are sharply decreased. If, for example, 5 percent of the code being worked on were interpreted and 95 percent compiled, then the overall program would run about twice as slow. Thus, one pays only a small price for the productivity benefits of this more interactive and responsive system.

`Cin` executes from flow sets, which are combinations of basic blocks with transfer-of-control instructions. The symbol table for the loader and preprocessor, along with the C-language identifiers, is kept in memory to allow expressions to be entered and interpreted in the appropriate context.

A source code browser. The C interpreter helps programmers understand a module's operation during development or maintenance. An interactive source-code browser, `samuel`, extends that ability, allowing them to investigate intermodule and intramodule relationships. In this environment, programmers can create a window for viewing information about the network of functions that call or are called by a current function. Or, they can call up a window with the definition used in the current module but not defined in the module. `Samuel` combines the capabilities of the `sam` editor with C-language browsing capability, as found in `cscope`,[21] and includes other online support facilities.

As the Masterscope browser (for the Interlisp® language) and the Smalltalk™ language browser have done for their environments, `samuel` helps the developer unravel the complex networks of subroutine linkages that are typical of large software efforts. (Smalltalk and Interlisp are trademarks of Xerox Corporation.) `Samuel`, like `cscope`, provides such information as where variables are used, where functions are defined, and what functions call and are called by a given procedure. In addition, `samuel` lets developers view multiple code segments simultaneously, tag important information, and easily access results of previous searches through menu items. Therefore, it is easy to trace a variable's effect on the program through several levels of procedure calls.

`Samuel` is especially useful for maintaining and debugging massive programs. It is also useful for walking through on-line code, traversing complicated file organizations, integrating voluminous code, coordinating work of programming teams, and training. Although a preliminary version has been written in C++ language, work will be done to handle the richer functionality of the language.

Research extensions. These are some of the current extensions and new directions:

- Investigate capabilities to support debugging and program visualization.
- Closely integrate the interpreter, browser, incremental debugging, and visualization facilities.
- Integrate "hypertext" relationships with the existing browser functionality in a distributed environment. (Hypertext refers to nonlinear documentation systems.)

An Environment for Text Creation. In most large systems, at least as much effort is expended on documents (requirements, design, documentation, etc.) as on code. Thus, advances in text processing lead to productivity gains for the overall software development process. Two critical issues for large software development projects are: how to generate the pieces of documents efficiently, and how to relate them to each other and to the software. Notice that these requirements directly parallel those for code. An additional need is for developers to relate code to relevant documentation, which will help keep requirements and design documents up to date with code changes.

Document preparation. An ongoing program of research is aimed at simplifying document creation, automating layout, and discovering the proper integration of interactive and batch publishing systems. An earlier fruit of this work is a database-driven document formatter, called `monk`.[22]

A prototype program[23] has been built for constructing tables interactively by direct visual means, not verbal specifications. Because this WYSIWYG (what you see is what you get) table editor also reads and writes `tbl`, it integrates with existing UNIX system software. A writer can read in or type new data into an existing template, or create the table from scratch. As characters are entered, the table expands but retains the desired alignment. As with `tbl`, text can span multiple columns or rows. Using a mouse and menus, the writer can add or delete rows and columns or cut, paste, and copy entire subtables. The surrounding context adjusts "intelligently" to include the new information.

Other work addresses the need to isolate and implement "galleying" primitives to produce high-quality documents. These primitives permit the writer or document designer to place pictures, tables, bitmaps, equations, and secondary text automatically; flow text into shapes or around oddly shaped artwork; and adjust vertical spacing to avoid problem pages in documents. C-language code can be mixed with macros, making complicated text processing algorithms easier to express and faster to run. (Macros are codes that tell UNIX system formatters how to process the document's material.) Also, the C-language routines can be modified at run time using the incremental loader in Cens.

Text and code relationships. The problem of maintaining the relationships of documents from various phases of the software life cycle is fundamental to managing development in large systems. As Figure 8 suggests, text creation dominates the early stages of a software system's life cycle, and code-related activities dominate later stages. Typically, the documents run to thousands of pages and often contain more lines than the source code. But if we use current methods, there is no feedback across the dashed line in Figure 8. Thus, if the documentation is not repaired, it starts to become obsolete when coding begins.

We can consider the set of units of documentation and code as a set of nodes that are connected by links and can be shared among documents. Each document is a collection of these nodes and links, organized nonlinearly in a computer, rather than linearly as on a printed page. Changes to any piece of this document and code network should propagate through the network so that the entire entity reflects a consistent state.

Current work is focused on the design and implementation of a demonstration hypertext shell that can be used to write a range of applications. Questions that are being studied in the context of these applications include:

- How can naturally related materials be linked automatically?
- How can this medium impose consistency when parts of a system are changed?

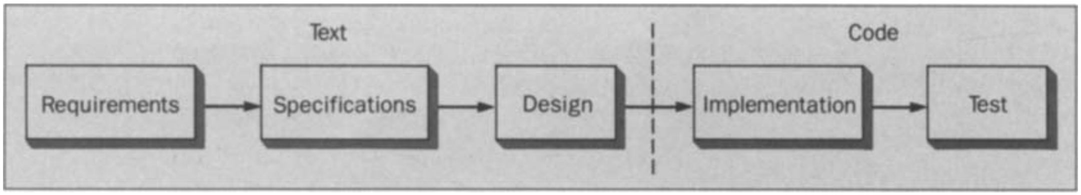


Figure 8. Text and code in the software life cycle. Feedback on changes currently does not cross the dashed line.

- What browsing aids are needed to navigate through such a complex network?

The long-term goal of our research on hypertext systems is to give developers an integrated view of all documents, notes, comments, and source code. This should improve understanding and encourage continued use of the documents throughout the software development cycle.

Summary

We have described a collection of research efforts aimed at improving the technology for large-scale software development. They include experiments with tools that support creation of a software system's individual units, automatically generate systems' platforms on which to build applications, and provide environments that manage a system's pieceparts as it evolves. These efforts represent a variety of approaches for overcoming obstacles to developing high-quality software more productively. They also illustrate some of the problems that must be addressed when building a large software system.

We divided this paper into unit, system, and product environments to reinforce the understanding that improvements *must* come from several directions. Because these tools will address problems caused by software's size and complexity, they are qualitatively different from those an individual programmer uses. That is, the tools themselves will be different from standard programming tools.

At a more abstract level, one can detect several common threads in the work we described:

- *Hidden complexity*—Several of these projects try to manage automatically some of the complexity of large system development, so the project team's members need not be aware of it. Examples are work on the Advanced Software Development Environment and `inscape`.
- *Increased context*—Any change to a large software product is likely to affect other parts of the product. Tools such as Cens, `cia`, and, a little further out, the hypertext work will give software developers immediate information about the way their work relates to the rest of the system.
- *Generation*—The generation of error code with `rip` and requirements with `watson` simplifies or eliminates parts of the development cycle, condensing its length.
- *Higher level platforms*—Platforms provide specialized services to software systems in an easy to use, highly tuned way. The platform for an application can control the productivity and quality of the resulting product. We described two platforms, Pegasus and Counterpoint.
- *Analysis*—To evolve into an engineering discipline, software development must analyze its products quantitatively and structurally. It must also provide a feedback loop so that these analyses affect ongoing design and development. The work in `cia` and `inscape` on analyzing software structure, as well as the work on visualization, leads in that direction. Often, the ability to analyze implies additional structure or formalism in the languages or processes used to develop applications.

The work we described builds on a long list of systems that originated as research prototypes and substantially affected the way software is developed at AT&T and elsewhere. Several of these systems are described in this issue. Some research efforts we described will also yield tools that will become standard elements of AT&T's development environment. In each case, the new prototypes will be subjected to a rigorous process of technology transfer.[24]

Acknowledgments

We wish to acknowledge the important contributions of Emden Gansner, Ted Kowalski, and Dewayne Perry to this paper, and also thank all researchers who did the work we described.

References

1. F. P. Brooks, Jr., *The Mythical Man Month*, Addison-Wesley, Reading, Massachusetts, 1978.
2. D. Korn and E. Krell, "Transparent Version Control for the UNIX Systems," *VIII International Conference on Computer Science*, Santiago, Chile, July 4-8, 1988, pp. 45-50.
3. V. B. Erickson and J. F. Pellegrin, "Build—A Software Construction Tool," *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 6, Part 2, July-August 1984, pp. 1049-1059.
4. S. Cichinski and G. S. Fowler, "Product Administration Through SABLE and NMAKE," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 59-70.
5. Y.-F. Chen and C. V. Ramamoorthy, "The C Information Abstractor," *Tenth International Computer Software and Applications Conference (COMPSAC)*, Chicago, Illinois, October 1986, pp. 291-298.
6. J. O. Coplien, S. C. Dewhurst, and A. R. Koenig, "C++: Evolving Toward A More Powerful Language," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 19-32.
7. N. Gehani and W. Roome, "Concurrent C," *Software Practice and Experience*, Vol. 16, No. 9, September 1986, pp. 821-844.
8. E. R. Gansner, S. C. North, and K.-P. Vo, "DAG—A Program that Draws Directed Graphs," *Software Practice and Experience*, to be published, 1988.
9. G. D. Bergland and P. Zave, "Prologue: Special Issue on Software Design Methods," *IEEE Transactions on Software Engineering*, 1986, pp. 186-191.
10. K.-P. Vo, "IFS—A Tool to Build Integrated, Interactive Application Software," *AT&T Technical Journal*, Vol. 64, No. 9, November 1985, pp. 2097-2117.
11. P. Zave and W. Schell, "Salient features of an executable specification language and its environment," *IEEE Transactions on Software Engineering*, Vol. XII, February 1986, pp. 312-325.
12. D. E. Perry, "The INSCAPE Environment: Knowledge-Based Synthesis of Large Systems through the Evolution of Program Interfaces," *AAAI Workshop on Automatic Programming*, Philadelphia, Pennsylvania, August 1986.
13. J. H. Reppy and E. R. Gansner, "A Foundation for Programming Environments," *Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, Palo Alto, California, December 1986, pp. 218-227.
14. C. D. Blewett, "Counterpoint: A Reasoning Based Graphics Toolkit," presented at MIT X Graphics Conference, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1988.
15. V. E. Kelly and U. Nonnenmann, "Inferring Formal Software Specifications from Episodic Descriptions," *Proceedings of the AAAI*, Vol. 1, July 1987, pp. 127-132.
16. V. E. Kelly and D. L. McGuinnness, "Automatic Re-Programming for Robustness," *Globecom-86*, Vol. 1, Houston, Texas, December 1986, pp. 0417-0422.
17. D. E. Perry and W. M. Evangelist, "An Empirical Study of Software Interface Faults—An Update," *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, Vol. II, January 1987, pp. 113-126.
18. C. D. Blewett, M. Wish, and J. I. Helfman, "A New IPC System for Bitmap Graphics Applications: Review, Model, and Benchmarks," *USENIX Summer Conference Proceedings*, Phoenix, Arizona, June 8-12, 1987, pp. 159-184.
19. D. McAllester, "Reasoning Utility Package User's Manual Version One," AI Memo 667, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1982.
20. J. C. Cleaveland and C. M. R. Kintala, "Tools for Building Application Generators," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 46-59.
21. J. L. Steffen, "Interactive Examination of a C Program with Cscope," *USENIX Winter Conference Proceedings*, Dallas, Texas, January 23-25, 1985, pp. 170-175.
22. S. L. Murrel and T. J. Kowalski, "Monk: A Database Driven Text Processor," *AT&T Technical Journal*, to be published, 1989.
23. S. L. Murrel and D. De Baer, "An Interactive WYSIWYG Table Editor," *USENIX Summer Conference Proceedings*, Phoenix, Arizona, June 8-12, 1987, pp. 19-29.
24. D. G. Belanger, "TECHNOLOGY TRANSFER: A Supplier View," *2nd IEEE Workshop on Software Technology Transfer*, Albuquerque, New Mexico, June 1987.

Biographies (continued)

systems. He joined the company in 1966 and has a B.S., M.S., and Ph.D. in electrical engineering from Iowa State University. Mr. Wish, head of the Computer-Aided Information Systems Research Department, joined AT&T in 1967 and specializes in research on computing environments, user interface design, and software productivity. He holds a B.S. in psychology from Case Western Reserve University, an M.S. and Ph.D. in mathematical psychology from the University of Michigan, and an M.S. in computer science from Columbia University.

(Manuscript received April 26, 1988)